TRATAMIENTO DE IMÁGENES USANDO EL MÉTODO DE PROFUNDIDAD DE CAMPO EXTENDIDA (EDF) EN ARQUITECTURAS PARALELAS

MONICA LILIANA HERNANDEZ ARIZA

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERIAS FISICOMECANICAS
ESCUELA DE INGENIERIA DE SISTEMAS
BUCARAMANGA
2011

TRATAMIENTO DE IMÁGENES USANDO EL MÉTODO DE PROFUNDIDAD DE CAMPO EXTENDIDA (EDF) EN ARQUITECTURAS PARALELAS

Autor:
MONICA LILIANA HERNANDEZ ARIZA

Proyecto de Grado para optar el título de Ingeniero de Sistemas

Director:
Ph.D Arturo Plata

Codirector:
Ph.D Carlos Jaime Barrios Hernández

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERIAS FISICOMECANICAS
ESCUELA DE INGENIERIA DE SISTEMAS
BUCARAMANGA
2011

AGRADECIMIENTOS

Deseo expresar un especial agradecimiento a:

Laboratoire d'Informatique de Grenoble, con una especial mención a los profesores Bruno Raffin y Olivier Richard por su colaboración en la utilización de los recursos computacionales del LIG.

Al Servicio de Computo de Alto Rendimiento y Calculo Científico de la Universidad Industrial de Santander (UIS), por la cooperación y soporte en el uso de la plataforma GridUIS-2. http://grid.uis.edu.co.

Al Director del proyecto, Profesor Arturo Plata Gómez Ph.D por contribuir con sus conocimientos en física para el desarrollo de este proyecto, por su consejo y valiosa colaboración.

Al Codirector del proyecto, Carlos Jaime Barrios Hernández Ph.D por la confianza depositada en mí, por su compromiso con el desarrollo del proyecto, por compartir sus conocimientos y por su constante apoyo y motivación.

Al ingeniero Leonardo Camargo Forero, por el apoyo incondicional y acompañamiento en el desarrollo de este proyecto. A ti, por enseñarme a dar lo mejor de mí, por estar presente en los momentos importantes de mi vida, por tu comprensión y por tu cariño.

Al ingeniero Antonio Lobo por su apoyo y sus consejos.

A todos los integrantes del grupo de Supercomputación y cálculo científico de la UIS, SC3.

DEDICATORIA

A mi familia por el amor que me entregan día a día

A ti mamá, porque con tu ejemplo me has enseñado la importancia de luchar por mis sueños.

Mónica Liliana Hernández Ariza

CONTENIDO

Pág.

LISTA DE FIGURAS

Pág.

LISTA DE ANEXOS

Pág.

ABREVIACIONES

CPU	Unidad Central de Procesamiento – *(Central Processing Unit)*
GPU	Unidad de Procesamiento Grafico – *(Graphic Processing Unit)*
CUDA	Arquitectura Unificada de Dispositivos de Computo – *(Compute Unified Device Architecture)*
MPI	Interfaz de Paso de Mensajes – *(Message Passing Interface)*
UMA	Acceso Uniforme a Memoria -- *(Uniform Memory Access)*
SMP	Multiprocesadores Simétricos – *(symmetric multiprocessing)*
NUMA	Acceso No Uniforme a Memoria – *(Non-Uniform Memory Access)*
MPP	Procesadores Masivamente Paralelos – *(Massively Parallel Processor)*
EDF	Profundidad de Campo Extendida – *(Extended Depth of Field)*
ALU	Unidad Aritmético-Lógica.

GLOSARIO

MEMORIA DISTRIBUIDA: Es propia de arquitecturas que cuentan con varias CPU's separadas geográficamente, donde cada una cuenta con una memoria propia. Para que una CPU-1 pueda acceder a una posición de memoria de otra CPU-2, debe establecer una comunicación con esta última empleando paso de mensajes, este tipo de memoria es propia de arquitecturas como cluster.

MEMORIA COMPARTIDA: Es propia de arquitecturas que cuentan con varias CPU's con una memoria común, donde todas las CPU's pueden acceder simultáneamente a una misma posición de memoria. Es propia de equipos multicore y tarjetas gráficas, que poseen varias unidades de procesamiento incluidas en un mismo chip.

MULTICORE: Es un tipo de procesador que cuenta con dos o más núcleos *(cores)* de procesamiento. Equipos como computadores de escritorio y portátiles, poseen procesadores multicore.

MANYCORE: Arquitectura que se caracteriza por una elevada cantidad de núcleos *(cores),* en donde todos los cores comparten una memoria.

STACK DE IMÁGENES: *Consiste en un conjunto de imágenes del mismo objeto que se encuentran focalizadas en diferentes puntos del plano de observación. Al desplazar el plano del foco a través de la imagen con una distancia de focalización en orden de micrones es posible obtener imágenes donde el campo de observación estará en centésimas de micras, al realizar un empalme de todo el stack como resultado se obtendrá una imagen focalizada con un mayor grado de detalle y nitidez que una imagen donde el plano de foco sea igual al campo de observación* [1].

RESUMEN

TITULO: TRATAMIENTO DE IMÁGENES USANDO EL MÉTODO DE PROFUNDIDAD DE CAMPO EXTENDIDA (EDF) EN ARQUITECTURAS PARALELAS[1]

AUTOR: MÓNICA LILIANA HERNÁNDEZ ARIZA[2]

PALABRAS CLAVE: Método EDF, Arquitecturas HPC, Cluster de computadoras, GPUs.

DESCRIPCIÓN: Existen diferentes problemas que debido a su grado de complejidad generan un volumen de datos que requieren una capacidad de computo elevada para su procesamiento, por lo tanto, estos datos no pueden ser tratados en un equipo de cómputo convencional en un intervalo de tiempo aceptable.

Un ejemplo de ello, es el tratamiento de imágenes de alta resolución en donde una imagen individual puede contener más de 5 millones de pixeles, es decir, que un apilado de imágenes puede generar un volumen de datos en orden de Gigabytes, incluso de Terabytes.

En este tipo de problemas, no es suficiente emplear equipos de cómputo que posean mejores características hardware que las que posee un equipo de cómputo convencional. Es necesario emplear técnicas y tecnologías que permitan aprovechar de mejor manera los recursos con los que se cuenta.

En la actualidad existen diferentes métodos mediante los cuales se realiza tratamiento de imágenes. El método de Profundidad de Campo Extendida EDF, es un método artificial que permite obtener una imagen focalizada en todo el campo de observación y una imagen de topografía de un objeto a partir de un apilado de imágenes focalizadas en diferentes planos.

En el presente trabajo de investigación se desarrolló un algoritmo que permite hacer una implementación del método EDF en arquitecturas de alto rendimiento computacional HPC, como cluster de computadoras y unidades de procesamiento grafico GPUs. La implementación del algoritmo en paralelo permitió obtener un mejor rendimiento respecto al tiempo de procesamiento de un apilado de imágenes, en comparación con la implementación serial empleada en investigaciones anteriores.

[1] Trabajo de Grado en la Modalidad de Investigación

[2] Facultad de Ingenierías Físico Mecánicas. Escuela de Ingeniería de Sistemas e Informática. Director: Arturo Plata Gómez. Ph.D. Codirector: Carlos Jaime Barrios Hernández. Ph.D.

ABSTRACT

TITLE: IMAGE TREATMENT USING EXTENDED DEPTH OF FIELD IN PARALLEL ARCHITECTURES[3]

AUTHOR: MÓNICA LILIANA HERNÁNDEZ ARIZA[4]

KEYWORDS: Extended Depth of Field, HPC Architectures, Computer Cluster, GPUs

DESCRIPTION: There are high complexity problems that generate a data volume that requires a high computing capacity for their treatment; therefore, this data can't be treated in a conventional computer with an acceptable efficiency.

The high-resolution image treatment is one example of this kind of problems. A high-resolution image can have over five million pixels, thus a stack of images could generate a data volume in order of Gigabytes, even Terabytes.

In this kind of scenario, it's not enough to use high capacity computers with better hardware characteristics that a normal computer equipment. It is also necessary to implement software techniques and technologies for exploiting in a better fashion the available resources.

The Extended Dept of Field (EDF) method is an artificial method for obtaining a completely focalized image in the entire observation field and a topography image from an object by using a partially focalized stack of images.

This research work shows the development of a High Performance Computing EDF by using a Cluster of computers and Graphic Processing Units (GPU). The implementation of the parallel algorithm obtained a higher performance and efficiency in different tests for some stack of images over a previous serial version of the method in another works.

[3] Undergraduate final project, research modality

[4] Systems Engineering and Computer Science School, Director: Arturo Plata Gómez. Ph.D. Codirector: Carlos Jaime Barrios Hernández. Ph.D.

INTRODUCCIÓN

El trabajo que se presenta a continuación ofrece diferentes alternativas para soportar la investigación científica mediante el uso de tecnologías de la información, específicamente computación de alto rendimiento.

En la actualidad el uso de tecnologías de alto rendimiento computacional se ha convertido en una herramienta necesaria para soportar diversos campos de la investigación científica, los cuales requieren grandes capacidades computacionales para obtener resultados de manera eficaz y sobretodo de manera eficiente.

Sistemas como cluster de computadores y sistemas híbridos CPU-GPU ofrecen cada uno diferentes características que pueden ser aplicadas a múltiples problemas dependiendo de la naturaleza de estos y de las posibles soluciones plateadas para su correspondiente resolución.

El tratamiento de imágenes, un área fuertemente aplicada en la investigación científica es uno de estos clásicos problemas que se ven ampliamente beneficiados con el uso de este tipo de tecnologías HPC.

En este trabajo se presenta la implementación de un método de tratamiento de imágenes conocido como Profundidad de Campo Extendida en dos diferentes tipos de plataformas computacionales, un cluster de computadores y un sistema híbrido entre CPUs y GPUs. Ambos ofrecen mejoras sobre una implementación anteriormente serial, las cuales son observadas en los claros incrementos en el rendimiento, la eficiencia, el manejo de mayor cantidad de información siempre dentro de los parámetros de la calidad y la efectividad.

Posteriormente, se realizan comparaciones entre ambos tipos de

implementaciones y se observan incrementos considerables en el rendimiento cuando se utilizan tecnologías híbridas, en las cuales, dependiendo de la naturaleza del trabajo a tratar, teóricamente se esperan mejoras del orden de 6 a 12 veces el tiempo serial.

Estos sistemas híbridos pretenden aprovechar la totalidad de los recursos de procesamiento computacional mediante la utilización de tarjetas gráficas, las cuales son especialmente diseñadas para soportar operaciones matriciales en paralelo gracias al gran número de unidades aritmético lógicas con las que cuentan. Estas capacidades superiores para el procesamiento matricial en paralelo son precisamente la razón de que un problema de tratamiento de imágenes se vea tan beneficiado por el uso de este tipo de sistemas híbridos.

A continuación se presenta *Tratamiento de imágenes usando el método de profundidad de campo extendida (edf) en arquitecturas paralelas*.

1. OBJETIVOS

1.1. OBJETIVO GENERAL

Analizar y Desarrollar un algoritmo a partir del Método de Profundidad de Campo Extendida (EDF) para el tratamiento de imágenes microscópicas usando procesamiento en paralelo.

1.2. OBJETIVOS ESPECÍFICOS

- Estudiar lineamientos para el desarrollo de aplicaciones paralelas que permitan la implementación de algoritmos asociados al tratamiento de imágenes.

- Analizar y mejorar un algoritmo paralelo para la implementación del método de Profundidad de Campo Extendido que permita su procesamiento en arquitecturas paralelas.

- Analizar la eficiencia de la implementación del algoritmo paralelo, en dos tipos de arquitecturas: una maquina multicomputadora Cluster que permite el análisis de multiprocesamiento usando procesadores normales y un arreglo de GPU’s que permite el procesamiento usando Procesadores Gráficos.

- Adquirir y seleccionar mosaicos y conjuntos (stacks) de imágenes para la prueba del algoritmo de EDF.

- Obtener una imagen focalizada y una imagen topográfica que permita la visualización y renderización en tres dimensiones de imágenes microscópicas para fines científicos.

2. MARCO TEORICO

2.1. ARQUITECTURAS PARALELAS

Las arquitecturas que soportan procesamiento en paralelo pueden clasificarse según la taxonomía de Flynn en diferentes grupos. Dependiendo la cantidad de instrucciones que conforman un algoritmo y de la cantidad de datos a tratar por dichas instrucciones, las arquitecturas paralelas se pueden clasificar en [1]:

SISD: *(Single Instruction Single Data),* este tipo de arquitectura cuenta con una única CPU, por tanto, por cada ciclo de reloj se ejecuta una sola instrucción sobre un solo dato.

SIMD: *(Single Instruction Multiple Data),* este tipo de arquitectura cuenta con varias CPU's, cada una de ellas ejecuta la misma instrucción en cada ciclo de reloj pero sobre datos diferentes. Un ejemplo de este tipo de arquitectura son las GPU's *(Graphic Processing Unit)*.

MISD: *(Multiple Instruction Single Data),* en este tipo de arquitecturas cada CPU ejecuta una instrucción diferente, pero sobre el mismo dato.

MIMD: *(Multiple Instruction Multiple Data),* en este tipo de arquitectura cada CPU puede ejecutar un grupo de instrucciones diferente sobre un conjunto de datos diferente. Dependiendo de las características hardware, una arquitectura MIMD se puede clasificar en:

Multiprocesadores: Cuenta con memoria compartida[5], este tipo de arquitecturas se clasifican en UMA (SMP) y NUMA.

[5]Ver Glosario: Memoria compartida

Multicomputadores: Cuenta con memoria distribuida[6], en este tipo de arquitecturas los datos son transmitidos a través de la red mediante la interfaz de paso de mensajes *(MPI-Message Passing Interface).* Los multicomputadores se clasifican en clusters y MPP.

El empleo de una arquitectura paralela permite mejorar el rendimiento de un algoritmo en términos del número de operaciones que realiza en un intervalo de tiempo. Una de las características con las que se desea contar en este tipo de arquitecturas es la escalabilidad, es decir, la capacidad que tiene una arquitectura para aprovechar hardware adicional sin tener que modificar un determinado algoritmo.

2.2. CLUSTER COMPUTACIONAL

Un cluster consiste en un conglomerado de equipos de cómputo convencionales que unidos soportan procesamiento en paralelo [1] [2]. Cada estación de trabajo que compone el cluster recibe el nombre de nodo, dependiendo del rol que desempeñe dentro del sistema, un nodo puede ser el nodo maestro o uno de los nodos esclavos, donde el maestro es el encargado de asignar a los esclavos las tareas que deben realizar. Un cluster puede ser clasificado por el tipo de recursos de sus nodos o por el tipo de servicio que ofrece.

2.2.1. Clasificación Según El Tipo De Recursos Que Posee

Según los recursos hardware y software que poseen los nodos, un cluster puede ser:

- *Cluster homogéneo:* este tipo de cluster se caracteriza por que cuentan con un hardware y un sistema operativo común para todos sus nodos.

[6]Ver Glosario: Memoria distribuida

- *Cluster heterogéneo:* este tipo de cluster se caracteriza por que cada uno de sus nodos puede contar con un hardware y un sistema operativo diferente al de los demás nodos.

2.2.2. Clasificación según el tipo de servicios que ofrece

Los motivos principales por los cuales se recurre a un cluster, además de su bajo costo y su fácil mantenimiento, es la necesidad de realizar cálculo y almacenamiento para grandes cantidades de datos que un equipo de escritorio convencional no podría soportar. Un cluster puede prestar diferentes tipos de servicios, como rendimiento, eficiencia, disponibilidad y escalabilidad. Según estos servicios un cluster se clasifica en:

Cluster de alto rendimiento: En este tipo de cluster se ejecutan problemas complejos que requieren de una gran capacidad de cómputo y de memoria.

Cluster de alta disponibilidad: Este tipo de cluster se caracterizan cada nodo cuenta con un nodo de respaldo en caso de que ocurra un fallo de hardware seguir funcionando y realizando las tareas.

2.3. PASO DE MENSAJES EN ARQUITECTURAS CLUSTER

El paso de mensajes es un paradigma de programación paralela, que permite establecer una comunicación entre arquitecturas de memoria distribuida[7]. El empleo de paso de mensajes, permite contar con una mayor eficiencia de los algoritmos sin sacrificar su portabilidad. El paso de mensajes fue estandarizado en un conjunto de rutinas que dan origen a la interfaz de paso de mensajes MPI (*Message Passing Interface*) [2].

[7] Aunque inicialmente estaba pensado para arquitecturas de memoria distribuida, hoy día hay implementaciones de pase de mensajes para arquitecturas de memoria compartida.

MPI cuenta con más de 100 funciones que incluyen diferentes tipos de parámetros, que ofrecen diversos beneficios a los usuarios [2] [3]. Dentro de los objetivos más importante de MPI se encuentra el poder mejorar el rendimiento de un algoritmo al establecer una comunicación eficiente entre los nodos de un cluster y la portabilidad del código a un cluster que cuenta con más nodos o cuando el código va a ser ejecutado en un ambiente heterogéneo.

Los diferentes nodos que componen el cluster estan en capacidad de ejecutar varios procesos en paralelo, debido a que cuentan con una arquitectura de procesador *multicore*[8]. Esta característica le permite a cada *core* del cluster ejecutar un proceso independiente a los demás, es decir, cada nodo puede ejecutar instrucciones diferentes sobre un grupo datos diferentes. Sin embargo, a cada *core* se le puede asignar más de un proceso a realizar dentro de una aplicación paralela, aunque los procesos adicionales que le son asignados estarán encolados o en estado de espera mientras el proceso que se está ejecutando finalice, dando una ilusión de paralelismo para un número n de procesos lanzados.

La forma básica de programación en un lenguaje incluyendo librerías de MPI, es definir un proceso como proceso maestro que estará encargado del envío de datos a los demás nodos y la posterior recepción, almacenamiento y disposición final de la información procesada. Los demás procesos se definen como procesos esclavos, en cada uno de ellos se puede definir los grupos de instrucciones que deben ejecutar sobre el grupo de datos que le son asignados.

2.4. SISTEMAS HIBRIDOS: CPU-GPU

Las GPU’s tradicionalmente se han empleado en el procesamiento de

[8] En español se conoce como multinucleo

gráficos de aplicaciones que estan relacionadas con simulaciones, videojuegos, aplicaciones 3D, entre otros. La principal característica de la GPU, es que cuenta con un gran número de ALU's destinadas a realizar procesamiento intensivo de datos *(ver figura 1)*, lo cual favorece el computo paralelo [4]. Gracias a esto, se ha dado un uso más ambicioso a la GPU que va más allá de los gráficos, actualmente la GPU's se emplean en resolver problemas de propósito general (GPGPU -*General Purpose GPU o GPU Computing)*, como aplicaciones científicas en las que se realiza un alto porcentaje de operaciones en coma flotante [5].

Con el propósito de reducir el trabajo en el procesador central y de aprovechar la capacidad computacional del procesador gráfico, se ha formado un sistema híbrido entre CPU y GPU. En este tipo de sistemas se cambia de un procesamiento centralizado a un coprocesamiento entre el procesador central y el procesador gráfico [6].

Como se puede observar en la figura 1, la GPU cuenta con un número superior de ALU's en comparación con la CPU, lo que le permite realizar una procesamiento intensivo de datos, convirtiendo a la GPU en una alternativa para realizar procesamiento de algoritmos que estan pensados para ser tratados en paralelo [7].

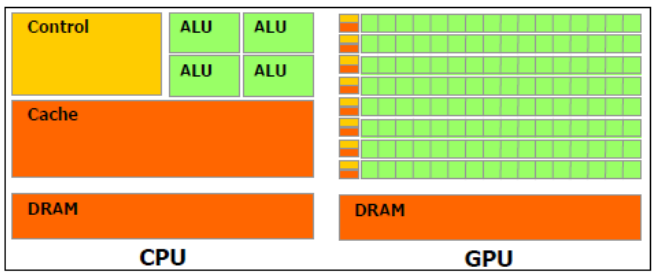


Figura 1. Imagen comparativa de la estructura de una CPU y una GPU. Tomado de NVIDIA CUDA compute Unified Device Architecture [7].

2.5. CUDA: ARQUITECTURA DE CALCULO PARALELO

CUDA® [8] es una arquitectura hardware y software creada por Nvidia® que posibilita la programación de algoritmos pensados para sistemas híbridos CPU-GPU en lenguajes de alto nivel como C y C++[9], donde la GPU se encarga de realizar aquellas partes del algoritmo que requieren mayor procesamiento y manejo de grandes volúmenes de datos [7] [8].

Un algoritmo programado para la arquitectura CUDA®, cuenta con funciones y variables que se declaran para la CPU, así como funciones y variables que se declaran para la GPU. En el contexto de CUDA®, se denomina *Host* a la CPU junto con la memoria RAM, *Device* a la GPU junto con su memoria. La función que es ejecutada por el *Device* se denomina *kernel* [8].

CUDA® permite programar el *Device* como una maya o Grid compuesta por múltiples bloques donde cada bloque a su vez está compuesto por múltiples hilos (*Ver figura 2*). Cada ciclo de reloj, todos los hilos de la *Grid* estan procesando en paralelo un bloque de instrucciones sobre un conjunto diferente de datos, lo que representa un mejor rendimiento del algoritmo y resultados obtenidos en un intervalo de tiempo inferior a los que se esperaría obtener si el algoritmo se programa de manera serial [8]. La forma como se distribuyen los bloques e hilos en la grid, facilita el tratamiento de problemas que implican vectores y matrices, permitiendo el procesamiento en paralelo de los mismos.

[9]En la actualidad, es posible la programación de CUDA en lenguajes como Java, Python, Fortran, entre otros.

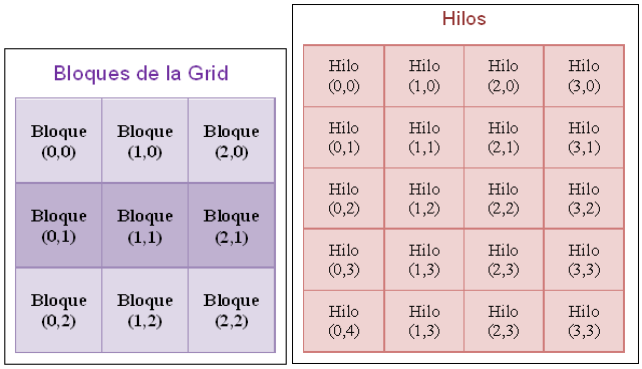


Figura 2. Dimensión de la Grid y de cada uno de los Bloques que la conforman.
Fuente: Autor

En la figura 2, se muestra un ejemplo de una Grid con dimensiones 3x3 bloques, donde cada bloque tiene una dimensión de 4x5 hilos. Como se observa en la figura, cada bloque e hilo tiene un identificador según su coordenada en el plano (x,y) en la Grid y en el bloque respectivamente. A diferencia de la notación algebraica que se emplea para numerar los elementos de una matriz, en CUDA® los bloques e hilos se numeran como se haría una matriz traspuesta, es decir, la primera coordenada corresponde a la columna y la segunda coordenada corresponde a la fila en la que está.

CUDA® es muy flexible en cuando a la forma en que se define el tamaño, es decir, si para la ejecución de un determinado código el programador determina que se emplearan *n* hilos, es posible obtener la misma cantidad de hilos mediante diferentes combinaciones de número de bloques e hilos por bloque. La forma como estan distribuidos los hilos y bloques en la Grid está ligada al diseño de un determinado algoritmo (*Ver figura 3*).

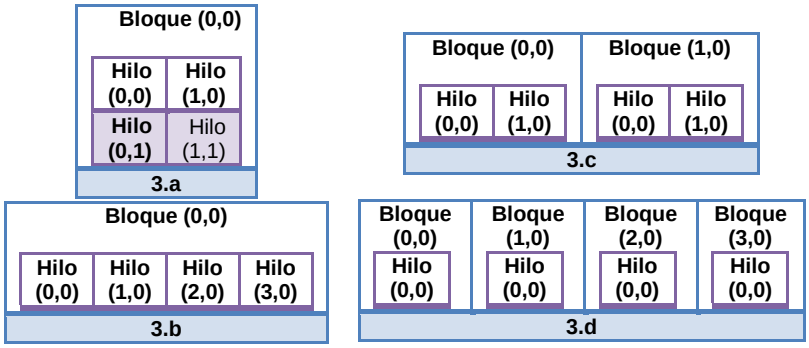


Figura 3. Diferentes tamaños para una Grid con 4 hilos. Fuente: Autor

En la figura 3, se muestran cuatro casos en los que se define una Grid con una cantidad de hilos igual a cuatro, según las dimensiones de la Grid, las coordenadas de los hilos y de los bloques cambian. En la figura 3.a y 3.b, se muestra una Grid de un bloque de 4 hilos distribuidos de manera diferente, en el primer caso como una matriz de 2x2 y en el segundo caso como un vector de 4 posiciones o una matriz 4x1. En la figura 3.c se muestra una Grid con 2 bloques de 2 hilos cada uno, la figura 3.d se muestra 4 bloques cada uno de 1 hilo.

Una Grid también puede definirse como un único bloque compuesto por múltiples hilos. En esta distribución la posición de un hilo ya no se determina por su posición en un bloque, sino por la posición del hilo en la Grid. Esta posición puede estar dada en dos dimensiones (x,y) o en una sola dimensión, dependiendo si la Grid se está tratando como una matriz o como un vector respectivamente [8].

Al hacer una comparación entre las primeras versiones de GPU's con las actuales, se observa la tendencia a aumentar el número de *cores* contenidos dentro de las tarjetas gráficas. Esto se debe al creciente interés hacia los beneficios que ofrecen arquitecturas como CUDA® en cálculos masivamente paralelos [7].

2.6. TRATAMIENTO DE IMÁGENES DIGITALES

El tratamiento de imágenes digitales consiste en un conjunto de técnicas que apoyadas en herramientas hardware y software permiten mejorar la calidad de una imagen, eliminando ruido, detectando y resaltando bordes, detectando intensidades, suavizando la imagen, etc. Estas técnicas suelen usarse para extraer información de interés a partir de una imagen [9].

2.7. PROFUNDIDAD DE CAMPO

La profundidad de campo es un concepto empleado en óptica y fotografía para hacer referencia al grado de nitidez que posee la imagen[10] de un objeto, organismo o paisaje en general. La profundidad de campo corresponde a la distancia que está por delante y por detrás del plano de focalización, donde se pueden observar detalles finos del objeto real. Cuando la borrosidad se hace mínima[11] en la profundidad de campo, se considera que la imagen es nítida en esa zona. La distancia nítida por detrás del plano de focalización es el doble que por delante de este [10].

La profundidad de campo se ve afectada por la apertura del diafragma, la distancia focal y por la distancia entre el plano de foco respecto al plano a focalizar.

- *Apertura del diafragma:* El diafragma es el que determina la cantidad de luz que pasa a través del lente de la cámara o microscopio. El diafragma puede graduarse de acuerdo al objeto observado. A medida que la apertura del diafragma decrece la profundidad de campo aumenta. Si la apertura del diafragma es muy grande, habrá una sobreexposición del

[10] Por imagen se hace referencia a fotografías, películas, etc.
[11] El grado de borrosidad o nitidez es relativo al observador.

lente a la luz, la profundidad de campo se reduce y el objeto se tornara borroso.

- *Distancia focal (zoom):* Es la distancia que hay desde el plano del lente de la cámara o microscopio respecto al plano de la imagen[12]. Al aumentar la distancia focal (acercar la imagen) la profundidad de campo disminuye, por ende los objetos fuera de la profundidad de campo se verán borrosos. Al disminuir la distancia focal (alejar la imagen) la profundidad de campo aumenta.

- *Distancia entre el plano de foco respecto al plano a focalizar:* Esta distancia hace referencia a cuan cerca está el lente del objeto. Al acercar el lente al objeto la profundidad de campo disminuye, los objetos o planos más cercanos al lente se desenfocan mientras que los planos más lejanos estarán focalizados. A medida que se aumenta la distancia del lente respecto al objeto la profundidad de campo aumenta.

2.8. METODO DE PROFUNDIDAD DE CAMPO EXTENDIDA – EDF

En óptica se emplean tecnologías como el microscopio óptico para obtener imágenes con resoluciones de decimas de micras sobre un área que es menor que el campo de observación. Estas imágenes se caracterizan por tener una profundidad de campo pequeña. Debido a que en cada imagen solo está focalizada un área pequeña, es necesario obtener un stack de imágenes[13] del mismo objeto, donde cada imagen estará focalizada en diferentes áreas [10].

La forma como se capturan las imágenes del stack en un microscopio óptico, es manteniendo constante los planos *x* e *y*, haciendo un desplazamiento en el plano *z* del objeto en orden de micrones, de manera que en cada captura

[12] El plano de la imagen es una superficie al interior de la cámara en donde se concentran los rayos de luz y se obtiene una imagen enfocada. Se mide en milímetros (mm).
[13] Ver Glosario: Stack de Imágenes

se obtiene un plano diferente focalizado. La cantidad de imágenes que contiene el stack depende del tamaño de objeto y del tamaño del paso al que se realiza el desplazamiento en el plano *z*.

Si se hace una comparación entre todas las imágenes del stack respecto a un mismo plano, se encuentra que las variaciones en la intensidades de color para la imagen en la que dicho plano se encuentra focalizado son mayores que para las imágenes que no lo estan. Por ende, es necesario emplear métodos que permitan localizar los planos focalizados a lo largo de cada imagen y a lo largo del stack.

El método de profundidad de campo extendida (*Extended Depth of Field*) es un método artificial que permite a partir de un apilado de imágenes obtener una imagen focalizada en todos los planos. Esto es posible, obteniendo la intensidad de color en la posición focalizada y la posición en la cual logro focalizarse para cada una de las imágenes, obteniendo una imagen focalizada y una imagen de topografía respectivamente [11]. Para lograr esto, el método EDF utiliza diferentes operadores como Varianzas, Sobel, Wavelets y Complex Wavelets.

A partir de la imagen de relieve o topográfica y la imagen focalizada, se reconstruye una imagen renderizada, que se caracteriza por dar una sensación de tridimensionalidad.

3. ESTADO DEL ARTE

En investigaciones anteriores se han realizado proyectos relacionados con la creación de algoritmos para el tratamiento de imágenes. En este tipo de proyectos se busca obtener imágenes de mejor calidad a partir de una imagen inicial o extraer información que resulte útil. A continuación se nombran algunas de estas investigaciones y se hace una breve descripción de los temas que se trataron en ellas.

- **Parallel Image Processing Based on CUDA**

Este trabajo se hace una presentación de la arquitectura CUDA y sus características para programación de propósito general sobre tarjetas gráficas. Se exponen algunos de los resultados obtenidos al implementar diferentes técnicas de tratamiento de imágenes como detección de bordes, utilizando programación CUDA. Se comprueban incrementos en el rendimiento al realizar la comparación entre CPU y GPU del orden de 200x, dejando de lado los tiempos de transferencia de datos entre CPU y GPU [12].

- **Design and Performance Evaluation of Image Processing Algorithms on GPUs**

Se realizan comparaciones entre algoritmos para procesamiento de imágenes programados sobre GPU's y sobre CPU's. Se realiza una comparación explícita entre códigos en CUDA y un programa muy rápido desarrollado utilizando OpenMP[14] [13].

- **Aspects of Parallel Image Processing Algorithms and Structures**

Este trabajo describe algoritmos para el procesamiento de imágenes y

[14] http://openmp.org/wp/

reconocimiento de patrones en arquitecturas paralelas. Se analizan principalmente aspectos de interconexión de redes y características inherentes a la computación de alto rendimiento como el grado de granularidad [14].

- **Implementation of parallel algorithm "conveyer processing" for images processing by filter 'mean'**

Este trabajo realiza la implementación de un algoritmo para la aplicación del filtro media en tratamiento de imágenes mediante el uso de la interfaz de paso de mensajes MPI [15].

- **A software architecture for user transparent parallel image processing**

Se hace una presentación de una arquitectura software que permite programar a los usuarios aplicaciones para tratamiento de imágenes que son mapeados a arquitecturas homogéneas de memoria distribuida de manera transparente para el programador. En base a “patrones paralelizables” se realiza la distribución del procesamiento sobre la arquitectura planteada [16].

- **Parallel image processing with OpenMP**

Este trabajo consiste en el tratamiento de imágenes tridimensionales mediante el uso de tecnologías OpenMP. Obteniendo mejoras en el rendimiento mediante la lectura y procesamiento paralelo de las imágenes a tratar [17].

A continuación se muestran algunos trabajos acerca del método de profundidad extendida. Estas investigaciones previas consisten en la implementación de diversos algoritmos secuenciales, los cuales presentan resultados interesantes que pueden ser mejorados claramente mediante el uso de tecnologías alto

rendimiento como las que se presentan en el desarrollo de este proyecto de grado.

- **Extending the Depth of Field in Microscopy Through Curvelet-Based Frequency-Adaptive Image Fusion**

Este trabajo presenta la complejidad que aparece al aplicar técnicas de profundidad de campo a objetos tridimensionales y ofrece una solución mediante la fusión de imágenes tratadas con la transformada en frecuencia *Curvelet* [18].

- **Virtual Microscopy with Extended Depth of Field**

La microscopía Virtual consiste en la utilización de un sistema compuesto por un computador personal que pueda reemplazar de manera satisfactoria la utilización de un microscopio óptico. En este trabajo se presenta el empleo de un sistema de microscopía virtual implementando profundidad de campo extendida. Existen algunas ventajas en el uso de este tipo de sistemas virtuales entre las cuales se incluye una disminución importante en los costos de adquisición de imágenes frente al uso de microscopios tradicionales [19].

- **To Extend the Depth of Field Based on Image Processing**

Este trabajo presenta resultados de la implementación de dos métodos para tratamiento de imágenes. Estos métodos son fusión de imágenes y restauración [20].

- **Extended-Depth-of-Field Iris Recognition Using Unrestored Wavefront-Coded Imagery**

El diseño de sistemas biométricos con reconocimiento de Iris provee mejoras

considerables a los sistemas de seguridad tradicionales. La obtención de imágenes claras y focalizadas del Iris es una tarea complicada debido a que es necesaria una participación completa del usuario en cuestión. Por este motivo es necesario desarrollar técnicas que permitan obtener imágenes del Iris que satisfagan los requisitos de estos sistemas biométricos. Este trabajo presenta los resultados de la implementación de dos algoritmos de reconocimiento, *Daugman's iriscode* y *correlation-filter-based iris recognition [21]*.

Además de proyectos de investigación existen aplicaciones secuenciales que emplean el método de profundidad de campo extendida EDF para tratar imágenes digitales. Estas aplicaciones emplean técnicas como detección de intensidades, de bordes y suavizado. Un ejemplo de este tipo de aplicaciones es ImageJ y Fiji.

- **ImageJ**

Es una aplicación basada en Java que permite realizar procesamiento sobre un apilado de imágenes. Las imágenes pueden de 8, 16 o 32 bits y tener diferentes formatos como JPG, TIFF, etc. [22]. ImageJ emplea diversos pluggings para hacer análisis y procesamiento de imágenes [11].

- **Fiji (Fiji Is Just ImageJ)**

Fiji es una distribución de ImageJ que emplea diversos pluggins y bibliotecas que permiten facilitar al usuario las su uso, en comparación con el mismo ImageJ [11].

El método de profundidad de campo extendida EDF es un método de tratamiento de imágenes empleado en investigaciones y trabajos previos a este proyecto de grado. Sin embargo, no se encontró documentación oficial que sugiera que este método ha sido implementado en arquitecturas paralelas.

4. EDF EMPLEANDO ARQUITECTURAS PARALELAS

En el presente proyecto se plantea e implementa una nueva alternativa para tratamiento de imágenes empleando arquitecturas de alto rendimiento computacional. La solución que se propone permite hacer procesamiento de stack de imágenes en paralelo, optimizando los tiempos de obtención de resultados y permitiendo el procesamiento de stack con un número importante de imágenes de formatos grandes.

La microscopia en una rama de la física que estudia objetos que no son visibles por el ojo humano o que carecen de forma definida. Para esto se emplea un microscopio óptico que permite enfocar zonas del objeto estudiado pero sobre un área menor que el campo de observación. Este tipo de situaciones obliga al observador a hacer un análisis fraccionado del objeto pues no se tiene una idea completa del mismo. Las imágenes que se obtienen al utilizar el microscopio óptico se caracterizan por tener una profundidad de campo pequeña debido a que el foco tiene un margen de profundidad estrecho. Esto ocasiona que los planos que se encuentran por delante y por detrás del plano focalizado sean borrosos.

Para el desarrollo de este trabajo fue necesario recurrir al método de profundidad de campo extendida EDF para poder obtener una imagen focalizada en todo el campo de observación a partir de un stack de imágenes focalizadas en diferentes planos. Sin embargo, los cálculos necesarios para poder localizar un plano focalizado, requieren una capacidad de cómputo elevada, debido a que es un proceso que se debe repetir para cada píxel contenido en la imagen y a su vez, para cada imagen contenida en el stack.

El método EDF emplea diversos operadores que permiten tratar una imagen de acuerdo a la necesidad que se tiene y al tipo de información que se espera

extraer de ella. En este caso, se tenía que encontrar las variaciones de intensidad de color entre las imágenes para cada punto, puesto que cuando un plano está focalizado, los puntos que se encuentran en él tienen intensidades de color mayores. Por tal razón, se determinó que el operador de varianza era el idóneo para cumplir con el propósito planteado. Este operador calcula las variaciones de intensidad de un píxel teniendo en cuenta los píxeles vecinos.

4.1. CALCULO DE LAS MATRICES DE VARIANZAS

Los colores de un pixel son el resultado de combinar en diferentes porcentajes los colores rojo, verde y azul (de ahí el nombre de color RGB, por las siglas en ingles Reed (R), Green (G), Blue (B)). Estos porcentajes tienen un equivalente numérico, de manera que el color de un pixel, puede expresarse como una terna de números (R,G,B).

Los equivalentes numéricos de las componentes RGB de un pixel pueden ser de 8 bits, 16 bits, 32 bits, etc. La diferencia entre los formatos implica una mayor variedad de colores, puesto que se puede realizar una mayor cantidad de combinaciones de las 3 componentes. Si se tiene un formato de 8 bits, quiere decir que cada componente tendrá un valor mínimo de 0 y un valor máximo de 255. Donde cero significa ausencia del color y 255 significa la presencia del color en su máxima intensidad (*Ver figura 4*).

COLOR	EQUIVALENTE NUMERICO DE LA COMPONENTE:		
	R	G	B
Negro	0	0	0
Azul	0	0	255
Verde	0	255	0
Cyan	0	255	255
Rojo	255	0	0
Magenta	255	0	255
Amarillo	255	255	0
Blanco	255	255	255

Figura 4. Equivalente numérico de las componentes RGB de los colores básicos. Fuente: Autor

En la figura 4, se muestra el equivalente numérico de las componentes RGB para los colores en donde hay ausencia total o presencia máxima de los colores rojo, verde y/o azul.

Cuando se varía el porcentaje de una o varias de las componentes se obtienen una paleta de diferentes colores. Para el formato de 8 bits, al variar los porcentajes se obtiene una paleta con aproximadamente (2^8)*(2^8)*(2^8)=2^24=16’777.216 colores, en la figura 5 se muestran algunos de los colores que resultan de variar los porcentajes de las componentes RGB:

COLOR	**EQUIVALENTE NUMERICO DE LA COMPONENTE:**		
	R	G	B
Salmon	250	128	114
Rosado	255	192	203
Violeta	199	21	133
Naranja	255	165	0
Fucsia	251	15	192
Cyan oscuro	0	139	139
Marrón	165	42	42
Azul cielo	135	206	235

Figura 5. Equivalente numérico de diferentes combinaciones de las componentes RGB. Fuente: Autor

El equivalente numérico de una imagen de resolución mxn son 3 matrices de

tamaño mxn, cada matriz contiene las componentes R, G o B para cada pixel de la imagen en cada posición ij.
Una vez se han obtenido los equivalentes numéricos se pueden realizar sobre estos datos operaciones matemáticas para calcular donde están contenidos los valores máximos. Cuando hay un cambio en la intensidad de color de un pixel, la componente verde refleja mejor estas variaciones. Es por esto que los cálculos de variaciones es preferible realizarlos sobre la matrizG.

Para calcular las variaciones de intensidad se calcula la varianza en cada elemento de la matriz, teniendo en cuenta sus vecinos más próximos [12]. Para ello se define una máscara de tamaño *pxp,* siendo p un número impar. La máscara es una matriz donde el elemento ubicado en la posición ij, siendo i=j, es el elemento al cual se le va a calcular la varianza, los demás elementos de la máscara corresponden a los vecinos más próximos. El propósito de definir una máscara con un número de elementos impar, es para tener claro cuál es el elemento central, es decir, donde i=j.

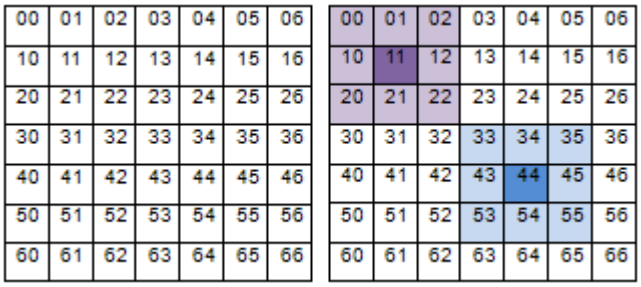


Figura 6. Desplazamiento de la máscara a lo largo y ancho de la matrizG. Fuente: Autor

En la figura 6 se observa un matriz de tamaño 7x7, suponiendo que esta matriz corresponde a la matrizG de una imagen con resolución 7x7 pixeles. Para realizar el cálculo de las varianzas se declaró una máscara de tamaño 3x3. La máscara se desplaza por todas las posiciones ij de la matriz, para determinar el punto en que se desea calcular la varianza y sus vecinos más cercanos. A

continuación se muestra la fórmula de varianza:

$$Varianza = \frac{1}{T} * \sum_{i=1}^{T} (Xprom - Xi)^2$$

$$Xprom = \frac{1}{T} * \sum_{i=1}^{T} Xi$$

Dónde:

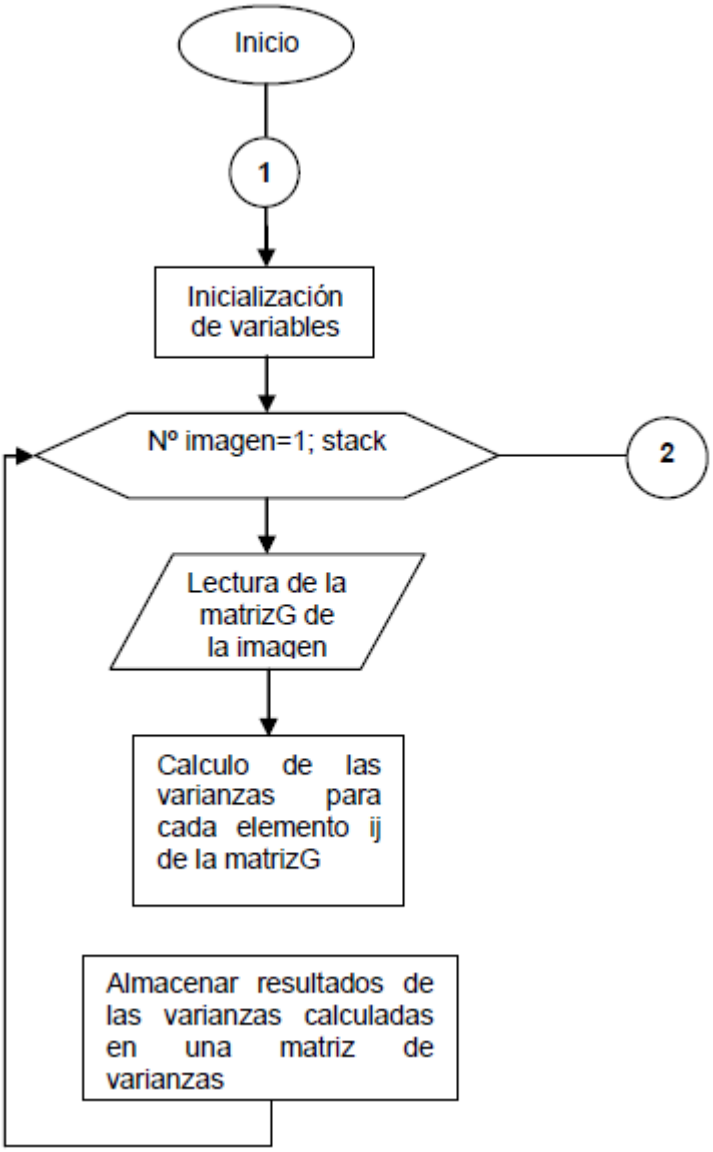
T es el tamaño de la máscara, es decir *pxp.*

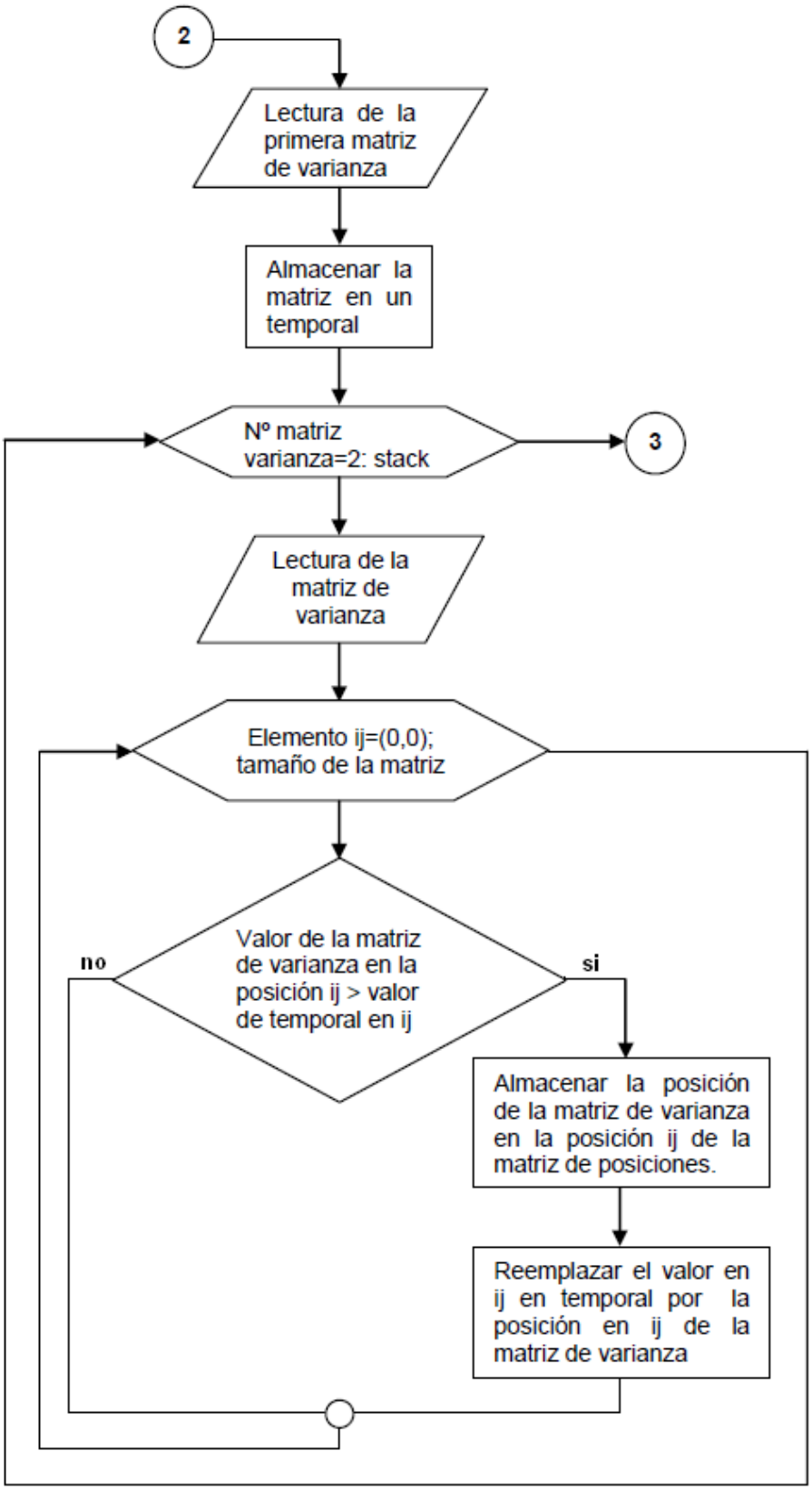
Xi es el valor numérico en la posición i

Xprom es el promedio de los elementos contenidos en la máscara.

4.2. EDF PARALELO

Debido la complejidad del método de profundidad de campo extendida y al volumen de datos que se generan a partir del tratamiento de imágenes, se propone la creación de un algoritmo y su procesamiento en arquitecturas paralelas. El algoritmo estará apoyado en el operador de varianzas para localizar las máximas intensidades de color en cada imagen del *stack*.


Inicio
1
Inicialización de variables
Nº imagen=1; stack
2
Lectura de la matrizG de la imagen
Calculo de las varianzas para cada elemento ij de la matrizG
Almacenar resultados de las varianzas calculadas en una matriz de varianzas


2
Lectura de la primera matriz de varianza
Almacenar la matriz en un temporal
Nº matriz varianza=2: stack
3
Lectura de la matriz de varianza
Elemento ij=(0,0); tamaño de la matriz
Valor de la matriz de varianza en la posición ij > valor de temporal en ij
no
si
Almacenar la posición de la matriz de varianza en la posición ij de la matriz de posiciones.
Reemplazar el valor en ij en temporal por la posición en ij de la matriz de varianza

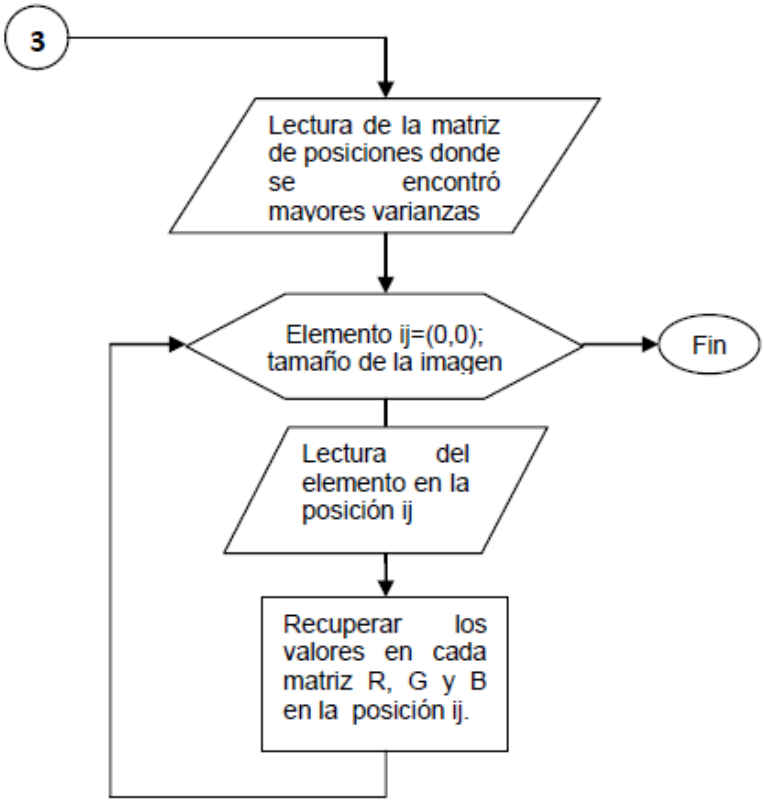


Figura 7. EDF Paralelo – Algoritmo propuesto. Fuente: Autor

En la figura 7 se muestra el diagrama de flujo del algoritmo propuesto para hacer tratamiento de imágenes empleando el método EDF con operadores de varianza. Como se observa en el diagrama, el algoritmo se dividió en 3 partes, como sigue:

1 Cálculo de las varianzas para las imágenes del stack.
2 Cálculo de la matriz de posiciones o de topografía.
3 Cálculo de las componentes RGB de la imagen focalizada.

En la primera parte, se calculan las variaciones de intensidad para cada una de las imágenes. Para ello se toman las matrices de la componente verde de cada imagen, pues este color es más sensible a los cambios de intensidad permitiendo determinar con mayor facilidad un punto de focalización.

Cada matriz de varianza posee un tamaño igual a la resolución de la imagen. Al finalizar esta parte del algoritmo se tiene una matriz de varianza por cada imagen en el stack, a partir de estos datos se hacen los cálculos necesarios para obtener la matriz de topografía.

En la segunda parte del algoritmo se hace una comparación punto a punto entre todas las matrices de varianzas con el fin de localizar los valores máximos para cada posición ij, una vez se encuentra este valor se almacena en la matriz de posiciones (topografía) la posición que ocupa dentro del stack la matriz donde está el máximo.

En la tercera parte del algoritmo se toma como referencias las posiciones de los máximos de intensidades para obtener las componentes RGB de cada punto focalizado. Es decir, se obtiene 3 matrices con los valores máximos para cada componente, al unir los valores en ij de cada matriz, se obtiene una terna que representa el color del pixel.

El algoritmo propuesto permite obtener una matriz de topografía que representa el relieve del objeto y 3 matrices de las componentes RGB que representa la textura del objeto. Al unir el relieve y la textura, se obtiene una imagen focalizada en 3 dimensiones.

4.3. IMPLEMENTACION DEL ALGORITMO

Para probar el desempeño del algoritmo, se plantea la implementación de dos tipos de arquitecturas paralelas: una arquitectura de memoria distribuida representada por un *cluster* de computadores y una arquitectura de memoria compartida representada por una GPU.

El algoritmo planteado está compuesto por secciones diseñadas para ser tratadas en paralelo. Sin embargo, el paralelismo está ligado a diversos factores, entre los cuales se encuentra el tipo de plataforma, la tecnología empleada y el tipo de arquitectura paralela. Esta dependencia permite que un algoritmo aproveche de mejor manera los recursos que posee una determinada arquitectura.

En la programación paralela es importante identificar según la arquitectura a emplear, que parte de un algoritmo pueden ser paralelizada, que partes no pueden serlo y cuáles no resulta conveniente paralelizar, puesto que en algunos casos un código paralelo puede incluso llegas a tener tiempos de procesamiento superiores a un código serial, debido a los costos de comunicación y la latencia característicos de una arquitectura paralela.

4.3.1. Cálculo de las componentes RGB de cada imagen

Para obtener las componentes RGB de las imágenes se creó un código en Matlab (*anexo A).* Este código lee cada imagen del stack, por cada imagen crea una carpeta cuyo nombre es la posición que ocupa la imagen en el stack. Cada carpeta contiene 3 archivos llamados R, G y B, que corresponden a las componentes Rojo, Verde y Azul de cada pixel respectivamente. Cada archivo esta expresado en forma de matriz, con un tamaño igual a la resolución de la imagen. Estos datos, son los datos de entrada al algoritmo y es a partir de ellos que se calculan las matrices de varianzas de cada imagen.

4.3.2. Cálculo de la matriz de varianza para cada imagen

Para realizar los cálculos de las varianzas en la intensidad de color alrededor de un punto, se definió una máscara con un tamaño de 3x3. Sin embargo, se presentó un problema en las esquinas y bordes, pues en estas partes no se contaba con los vecinos necesarios para hacer cálculos. Para solucionarlo, se agregaron 2 columnas y 2 filas a la matriz de la componente verde (matriz G) en cada una en uno de los extremos de la misma, a estos elementos

adicionales se les dio un valor numérico de cero.
Si se toma el ejemplo de la figura 6, se tiene:

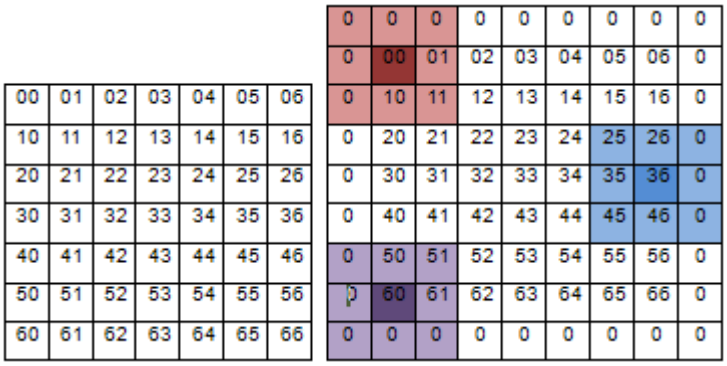


Figura 8. Modificación de la matriz G original para realizar cálculos de varianza. Fuente: Autor

En la figura 8 se muestra una matriz de la componente verde de una imagen de resolución 7x7 y una matriz aumentada en 2 filas y 2 columnas, para completar los vecinos de los puntos ubicados en los bordes y esquinas y de esta manera calcular las varianzas. Sin embargo el tamaño de las matrices de varianzas no se ve afectado, teniendo las mismas dimensiones que la matriz G original.

Debido a que las arquitecturas planteadas estan basadas en diferentes paradigmas de programación paralela, el cálculo de las varianzas empleando un cluster de computadores y una GPU se hace de diferente manera.

- *Cálculo de las matrices de varianzas empleando un cluster*

Para calcular las matrices de varianzas se creó un código en C que incluye librerías de MPI- *Message Passing Interface*. (Anexo B). Para esta parte del algoritmo se define un número *p* de procesos igual al número de imágenes que forman el stack, de esta manera se consigue que cada imagen sea procesada en paralelo.
En MPI, a los procesos les es asignado un identificador (id) en la medida que cada uno responde. El id=0 se le asigna al primer proceso en responder, el

id=p-1se le asigna al último. Es esta primera parte del algoritmo se define como proceso maestro a aquel proceso con id=0. Para este algoritmo, el proceso maestro no realizara cálculos, solo envió y recepción de datos. Luego, el maestro es el encargado de enviar a cada proceso esclavo una matriz G y recibir la información procesada por cada uno de ellos, para posteriormente almacenarla (*ver figura 9*).

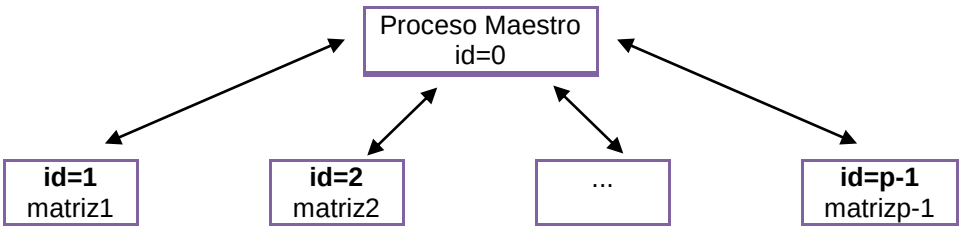


Figura 9. Envió y recepción de información entre procesos. Fuente: Autor

Como se observa en la figura 9, se establece una comunicación entre los procesos de tal manera que se permita un traspaso de información entre el proceso maestro con los demás. Para realizar envío y recepción de información entre procesos se utilizan las funciones *MPI_Send() y MPI_Recv() [2]*.

Cada proceso se encarga de calcular la máscara para cada elemento de la matriz que recibe. Para la máscara se define un vector M[i] de 9 posiciones. Siendo G la matriz que recibe el proceso, La máscara para cada elemento de la matriz G se define como sigue a continuación:

```
for(i=1; i<=dimx; i++)
                for(j=1; j<=dimy; j++){

                        M[0]=G[i-1][j-1];
                        M[1]=G[i-1][j];
                        M[2]=G[i-1][j+1];
                        M[3]=G[i][j-1];
                        M[4]=G[i][j];
                        M[5]=G[i][j+1];
                        M[6]=G[i+1][j-1];
                        M[7]=G[i+1][j];
                        M[8]=G[i+1][j+1];
                }
```

Figura 10. Declaración de la máscara para cada elemento. Fuente: Autor

En la figura 10 se muestra el segmento de código ejecutado por cada proceso, donde se define la máscara para cada elemento ij. Donde dimx es la dimensión de la matriz en x y dimy la dimensión de la matriz en y. M es la máscara y G la matriz G aumentada.

Para una imagen de resolución 640*480 píxeles, se tendría una matriz G de 640*480 elementos, la matriz G aumentada tendrá una dimensión de 642*482 elementos. El segmento de código que calcula la máscara está compuesto por dos *for* que permiten hacer un desplazamiento por la matriz en x e y. Según la figura 10, para una matriz de 642*482 elementos, las sentencias *for* inician en 1 para i y j, y finalizan en 640 para i que representa el desplazamiento en x y en 480 para j que representa el desplazamiento en y. Esto se hace con el fin de evitar que el proceso calcule la varianza para los elementos que se encuentran en las esquinas y bordes, como se observa en la figura 11.

El primer elemento al que se le calcula la máscara es al elemento en la posición 1,1. Una vez calculada la varianza de acuerdo a los valores del vector M, este valor se almacena en el elemento [i-1][j-1] de la matriz de varianzas, que para este caso sería la posición 0,0. Este proceso se repite para cada uno de los elementos de la matriz G.

0,0	0,1	0,2	0,3	...	...		0,639	0,340	0,641
1,0	1,1	1,2	1,3				1,639	1,640	1,641
2,0	2,1	2,2	2,3				2,639	2,640	2,641
...									...
...									...
									...
480,0	480,1	480,2					480, 639	480, 640	480, 641
481,0	481,1	481,2					481, 639	481, 640	481, 641

Figura 11. Calculo de la matriz de varianza en cada proceso. Fuente: Autor

- *Cálculo de las matrices de varianzas empleando una GPU*

A diferencia de cómo se paralelizo este segmento de algoritmo en un cluster, donde todas las matrices son procesadas en paralelo, para aprovechar los recursos de la GPU, cada matriz G es procesada de manera serial, pero las operaciones que se realizan a cada elemento de dicha matriz se hacen en paralelo.

Para esta sección del algoritmo se creó un código en C que incluye librerías de CUDA®. El *Host* se encarga de leer la matriz G y enviarla al *Device* donde se calcula la matriz de varianza, una vez calculada el *Device* envía de regreso los resultados al *Host* donde son almacenados[15], este proceso de comunicación se hace necesario debido a que el Device no realiza lectura de variables globales ni almacenamiento de datos (Anexo C).

[15]Ver Glosario: Host, Device.

En este código se definió un número de hilos igual al número de elementos que tiene la matriz G original, donde cada hilo se encarga de calcular la máscara de uno de los puntos de la matriz y la varianza para dicho punto teniendo en cuenta los vecinos que calculó, es decir, por cada llamado del *kernel* son desplegados un número k de hilos que se ejecutan en paralelo procesando una matriz de varianza.

Los hilos están distribuidos en diferentes bloques. Cada bloque de hilos procesa una fila de la matriz, es decir, cada bloque tiene un número de hilos igual al número de columnas de la matriz.

En CUDA® los hilos pueden estar identificados según su posición en ij dentro de la Grid. Sin embargo, para este caso específico se encontró que al cambiar las dimensiones de la matriz G, se presentaban problemas de desbordamiento de memoria debido a que el hilo intentaba acceder a un elemento ij de la matriz G que no existía. Debido a esto, se decidió una vez leída la matriz G por el *Host,* pasar estos valores al *Device* en forma de vector, de tal manera que cada hilo i se encarga de calcular el elemento i del vector de varianza.

Para el caso de una imagen con una resolución de 640x480 pixeles, la declaración de los hilos es como sigue:

Tamaño de la matriz G=640*480=307200 elementos
Número de hilos desplegados=640*480=307200 hilos
Tamaño de la matriz G aumentada (vector G)=642*482=309444 elementos

0	1	2	3	…	…	…	639	640	641
642	643 0	644 1	645 2	…	…	…	1281 638	1282 639	1283
1284	1285 640	1286 641	1287 142				1923 1278	1924 1279	1925
1926	1927 1280	1928 1281	1929 1282						…
…									…
308160	308161 306500							308800 307199	308801
308802	308803							309442	309443

Figura 12. Cálculo del vector de varianzas en el Device. Fuente: Autor

En la figura 12, se muestra el cálculo de las varianzas para cada elemento del vector G. los números de color azul hacen referencia a los elementos del vector G, los números en color rojo hacen referencia a los id de los hilos de la Grid.

Las filas y columnas que se encuentran en los extremos son aquellas filas y columnas que se agregaron al vector original para poder calcular las varianzas en los bordes y las esquinas. El objetivo de no desplegar un número de hilos igual al número de elementos del vector aumentado es no invocar recursos que no serán utilizados, pues los elementos adicionales solo sirven para completar la máscara más no se calcula varianzas en esos puntos.

Los elementos que estan sombreados de color lila, hacen referencia a los elementos del vector original, es decir, aquellos en los que se encuentran los valores de la componente verde de la imagen. Según la figura, en cada casilla se encuentra el id del hilo y el elemento del vector que será procesado por dicho hilo. Por ejemplo, el hilo con id=0 procesa el elemento 643 del vector, el hilo con id=139 procesa el elemento 1282 del vector. Los elementos de la

primera fila y la última no se tienen en cuenta pues son las columnas agregadas, por eso se procesa desde el elemento 643.

Nótese que el hilo 640 procesa el elemento 1285 en lugar del elemento 1283, que sería el elemento que debería procesar según el orden que llevaban los hilos iniciales. Algo similar pasa con el hilo 1280 que procesa el elemento 1927 en lugar del elemento 1925. Esto se debe a que hay 2 elementos que no deben ser procesados, los que estan en las columnas agregadas.

En conclusión, cuando el id del hilo es múltiplo de la dimensión en y de la imagen, el id debe aumentarse en 2, para no procesar un elemento de las columnas agregadas.

```
 offset+=2*idy;
 id_p=offset+(dimy+msk);

M_d[9];

M_d[0]=G_d[offset];
M_d[1]=G_d[offset+1];
M_d[2]=G_d[offset+2];
M_d[3]=G_d[id_p-1];
M_d[4]=G_d[id_p];
M_d[5]=G_d[id_p+1];
M_d[6]=G_d[id_p+dimy+1];
M_d[7]=G_d[id_p+dimy+2];
M_d[8]=G_d[id_p+dimy+3];
```

Figura 13. Declaración de la máscara en el *Device.* Fuente: Autor

En la figura 13 se muestra como se declara la máscara en el *Device.* Donde:
Offset es el id del hilo en el bloque.
Idy es el id de cada bloque, el idy del primer bloque es 0 y del último bloque es 479.
Id_p es el id del elemento a quien se le calcula la varianza

Dimy la dimensión de la matriz G en y
M_d es el vector de la máscara
G_d es el vector G

Para los hilos del bloque 0, el offset no cambia, pero para los hilos siguientes el offset se incrementa 2 veces el idy. El valor de varianza que se calcula para cada elemento se almacena en el vector de varianzas en la posición i, donde i es igual al id del hilo en la Grid.

4.3.3. Cálculo de la matriz de topografía

Una imagen de topografía es una imagen que permite apreciar el relieve de un objeto real. Para calcular la matriz de topografía se hace una comparación punto a punto entre las matrices de varianzas calculadas. De manera que donde se encuentre un máximo se almacena la posición de la imagen en el *stack.*

- *Calculo de la matriz de topografía empleando un cluster*

Debido a que comparar 2 valores y determinar cuál es el mayor, no requiere capacidad de cómputo, se definió que solo un proceso realizara esta sección del algoritmo, es decir, implementar el algoritmo en un código serial. Al emplear varios procesos se puede presentar un incremento en el tiempo de procesamiento debido al costo de comunicación y la latencia propia de un *cluster* (Anexo D).

Para esta parte del algoritmo se creó un código en C. La matriz de varianza de la imagen 1, es almacenada en una variable temporal. Después, se lee la siguiente matriz y se compara cada posición ij de las dos matrices. Si para una posición ij la matriz de varianza tiene un valor mayor que temporal, el valor en ij de temporal se actualiza y se almacena en la posición ij de la matriz de

topografía la posición que ocupa la matriz del máximo en el *stack*. Este proceso se repite para todas las matrices, ver figura 14.

```
for(i=0;i<dimx;i++){
        for(j=0;j<dimy;j++){
                if(varianza[i][j]>max[i][j]){
                        top[i][j]=k;
                        max[i][j]=varianza[i][j];
                }
        }
}
```

Figura 14. Calculo de la matriz de topografía empleando un solo proceso. Fuente: Autor

En la figura 14 se muestra la parte del código en la que se calculan los valores máximos presentes en cada matriz, donde dimx y dimy son las dimensiones de la matriz de varianza en x e y respectivamente, max[i][j] es la matriz que almacena los máximos a medida que se lee cada matriz de varianza, top[i][j] es la matriz de topografía.

- *Calculo de la matriz de topografía empleando una GPU*

Para este segmento de código se invocan un número de hilos igual al número de elementos de la matriz de la matriG. Cada matriz de varianza es enviada al *Device* como un vector con un número de elementos igual a dimx*dimy.

En el *Host* se leen los vectores que son enviados al *Device*, quien se encarga de calcular los valores máximos. Cada hilo compara el elemento i de los vectores que recibe y devuelve al *Host* los resultados. En la figura 15 se muestra el segmento de código ejecutado en el kernel que calcula la matriz de topografía, idx es el id del hilo en la Grid (*Anexo E*).

```
if(varianza[idx]>maximos[idx]){
                top[idx]=k;
                maximos[idx]=varianza[idx];
}
```

Figura 15. Calculo de la matriz de topografía en el *Device.* Fuente: Autor

Teniendo en cuenta que las operaciones que realiza cada hilo no son complejas, en este segmento de algoritmo se puede ver reflejado el costo de comunicación entre el *Host-Device-Host.* En situaciones donde las instrucciones que se ejecutan sobre un grupo de datos no tienen un grado de complejidad importante, es necesario determinar si se justifica emplear una arquitectura que soporte procesamiento en paralelo.

4.3.4. Calculo de las componentes RGB de la imagen focalizada

Las componentes RGB de la imagen focalizada se obtienen a partir de la matriz de topografía. Una imagen focalizada es una imagen de la textura permite apreciar los detalles más finos del objeto real. Una imagen de relieve y su imagen de textura permiten obtener una imagen 3D. Debido a que los cálculos necesarios para obtener las componentes RGB de la matriz focalizada requieren una capacidad de computo elevada, esta sección del algoritmo no se implementó en un cluster, pues debido al volumen de datos que hay que enviar a cada proceso a través de la red adicional al hecho de que se deben encolar cierta cantidad de procesos, el rendimiento del algoritmo se verá afectado. Identificar en la fase de diseño el tipo de situaciones en las que el rendimiento de un determinado algoritmo se puede ver afectado por la naturaleza misma de la arquitectura que se piensa emplear permite optimizar los resultados que se esperan obtener a partir de la implementación del mismo.

- *Calculo de las componentes RGB de la imagen focalizada empleando una GPU*

En esta parte del algoritmo se busca formar una matriz por cada una de las

componentes de una imagen. Cada matriz contiene por cada elemento, el máximo valor encontrado en el *stack* para un pixel. En la figura 16 se muestra el segmento de código ejecutado en el *kernel,* donde se forman las matrices R, G y B que representan los equivalentes numéricos de la imagen focalizada. Donde Rf, Gf, Bf son las componentes finales.

```
 if(T_d[idx]==d){
                    Rf[idx]=R_d[idx];
                     Gf[idx]=G_d[idx];
                    Bf[idx]=B_d[idx];
}
```

Figura 16. Calculo de las componentes RGB de la imagen focalizada en el kernel. Fuente: Autor

En cada llamado del *kernel*, el Host debe enviar al *Device* las matrices R, G y B de una imagen diferente, la matriz de topografía y las matrices R, G y B temporales, que son las que se forman elemento a elemento con cada ejecución del *kernel.* Debido a la cantidad de datos que deben ser enviados *Host-Device-Host* hay un costo de procesamiento elevado (Anexo F).

Cuando se paraleliza un algoritmo, es importante identificar que segmentos de este pueden ser tratados en paralelo dependiendo de la arquitectura que se emplea. Si se emplea un cluster para tratar esta parte del algoritmo, sería necesario enviar a cada proceso la misma cantidad de datos que se están enviando al *Device.* Por este motivo, no es conveniente recurrir a un cluster debido a la latencia propia del proceso de comunicación entre procesos y al volumen de datos que deben pasar por la red.

5. PRUEBAS Y ANALISIS DE RESULTADOS

Para la realización de las pruebas se utilizaron dos tipos de plataformas, empleando los recursos del Laboratorio de Supercomputación y Cálculo Científico de la UIS y empleando los recursos de la plataforma Grid5000[16].

- *Supercomputación y Cálculo Científico de la UIS*

Las pruebas realizadas en esta plataforma para el algoritmo planteado se llevaron a cabo en un *cluster* de computadoras y en una tarjeta GPU. El *cluster* utilizado está compuesto por 5 nodos de trabajo, con Sistema Operativo Red Hat 4.1.2-42, procesador Dual-core Pentium 4, CPU 3.20 GHz, memoria RAM 2GB. La tarjeta gráfica corresponde a una tarjeta Nvidia® Quadro® FX 570 con una memoria de 256MB, instalada en un equipo que posee las mismas características de los equipos que conforman el *cluster,* los equipos estan conectados a través de una red tipo ethernet.

PRUEBA Nº1: Esta prueba tuvo como objetivo analizar el rendimiento de la primera sección del algoritmo planteado para cálculo de las varianzas. Se realizaron unas pruebas iniciales con 19 matrices de dimensiones 640filas *480 columnas donde cada elemento de la matriz tiene un valor numérico de 1. Esto con el fin de corroborar la veracidad de los resultados. Estas matrices representan las matrices que se obtienen a partir de la componente verde de una imagen. Al calcular las varianzas de una matriz donde todos sus elementos son 1, la matriz de varianza debe ser como se muestra en la figura 17 (Anexo H).

[16] https://www.grid5000.fr

0.25	**0.22**	**0.22**	**...**	**...**	**0.22**	**0.22**	**0.25**
0.22	0	0	..	...	0	0	0.22
...	0	0	...	...	0	0	...
0.22	0	0	...	...	0	0	0.22
0.25	0.22	0.22	...	...	0.22	0.22	0.25

Figura 17. Matriz de varianza a partir de una matriz G donde cada elemento ij es 1.
Fuente: Autor

Al contar con un cluster de 5 nodos se tenían un total de 20 procesos en paralelo, uno por cada core. Como el código se planteo de manera que el proceso maestro realizara solo envío y recepción de datos mas no procesamiento de los mismos, se disponía de 19 procesos encargados de procesar una matriz por proceso. Si se asignan una cantidad mayor de procesos por core, los procesos restantes son encolados. Para la GPU, se invocaron un número de hilos igual a la cantidad de elementos de la matriz, en este caso 307200 hilos. Para analizar el comportamiento de las 2 arquitecturas para este tamaño de matriz, se incremento la cantidad de matrices a procesar en 1. A continuación se muestra en la figura 18 los tiempos de procesamiento empleados por cada arquitectura:

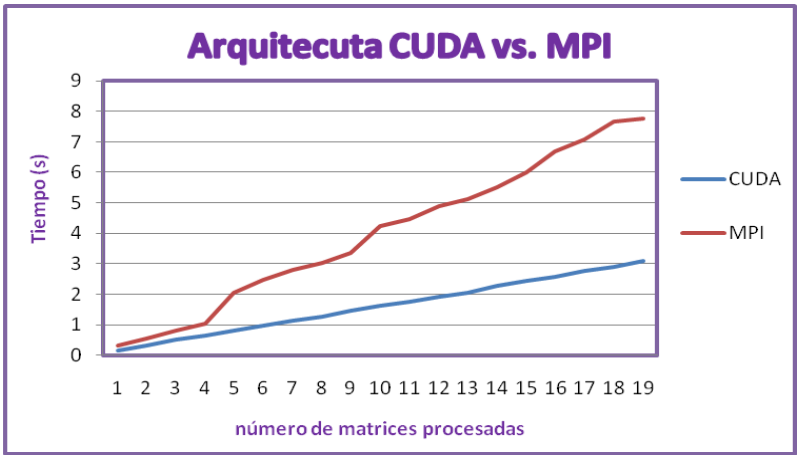


Figura 18. Tiempos de ejecución CUDA vs. MPI. Fuente: Autor

En la figura 18 se muestra el comportamiento de cada arquitectura al aumentar el número de matrices a procesar. Como se puede observar, se presenta un crecimiento lineal del tiempo de ejecución en la medida que se aumenta la cantidad de matrices procesadas. Es importante resaltar que existe una diferencia de un factor de 6, cuando la cantidad de matrices procesadas aumenta a 19. Esto se debe a las ventajas que tiene el procesamiento de problemas de grano fino en CUDA®, debido a que por un lado la arquitectura permite distribuir las matrices más eficientemente sobre la GPU y por otro lado el tiempo de comunicación es muy bajo, debido a la arquitectura misma de la tarjeta Quadro FX 570.

En la figura 19 se muestra los resultados del tiempo promedio de procesamiento por cada matriz a medida que se aumenta la cantidad.

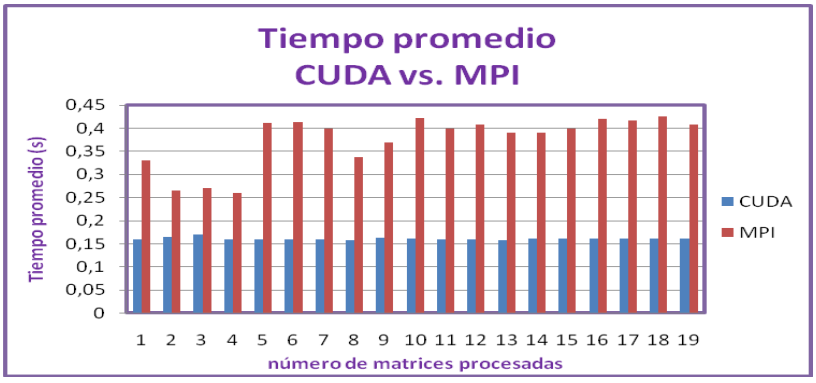


Figura 19. Tiempos promedios por matriz CUDA vs. MPI. Fuente: Autor

En la figura 19 se presenta una comparación del tiempo promedio que emplea el algoritmo en CUDA® y en MPI, como resultado de dividir los tiempos de ejecución (figura 18.) entre el número de matrices procesadas. Como se observa en la figura, el tiempo promedio en CUDA tiende a mantenerse estable al aumentar la cantidad de matrices, lo que no ocurre con MPI, donde se muestra una inestabilidad en los tiempos promedios. La diferencia que se observa sin duda es debido a los costos de comunicación generados en MPI debido al uso de las barreras que emplea y al alto costo de comunicación entre procesos [23].

PRUEBA Nº2: Esta prueba tuvo como objetivo analizar el rendimiento de la segunda seccion del algortimo planteado para calcular la matriz de topografia. Para esta parte, en el cluster le fue asignado a un solo proceso la tarea de hacer las comparaciones punto a punto de cada matriz, de manera que el rendimiento del codigo no se viera afectado por los costos de comunicación entre procesos. Para el codigo en CUDA, se invocaron un numero de hilos igual a 307200, un hilo por cada elemento de la matriz. Para esta prueba se tienen como datos de entrada las matrices de varianza calculadas a partir de la prueba nº1.En esta prueba se pretende averuguar si se justifica el empleo de una GPU o un cluster para tratar este problema especifico, partiendo del hecho que el problema en si, no requiere una capacidad computacional elevada.

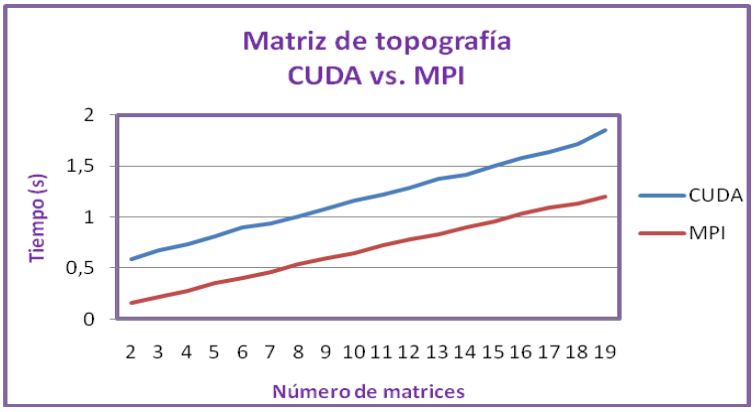


Figura 20. Calculo de la matiz de topografía en CUDA vs. MPI. Fuente: Autor

En la figura 20 se puede observar que se presenta un crecimiento lineal al aumentar el numero de matrices. Se puede apreciar el hecho de que los tiempos de procesamiento en los dos tipos de arquitecturas son bajos, no superando los 2 segundo para procesar un numero total de 19 matrices. Sin embargo, existe una diferencia de un factor de 0.5 entre el codigo en CUDA y MPI.

Debido a que el codigo en MPI es ejecutado de manera serial, pues solo se emplea un proceso para realizar los calculos, el tiempo no se ve afectado por los costos en comunicación y la latencia propia de un cluster. En el caso de CUDA, a pesar de que los tiempos son bajos, se ve reflejado el costo en la comunicación entre el *Host* y el *Device.* Esta comunicación es necesaria pues el Device no puede almacenar datos en su memoria.

PRUEBA Nº3: Esta prueba se realizo a partir de un stack de 50 imágenes con un formato TIFF. La resolucion de una imagen de este formtato es de 1920*2560 pixeles. Para obtener las componentes RGB se utilizo el código en matlab que se muestraen en el Anexo A. En esta prueba se obtuvieron las matrices de varianza de cada imagen a partir de la matriz de la componente verde de cada una de ellas.

Para la ejecucion del codigo en MPI se emplearon un numero de procesos igual a 50, es decir, un proceso por imagen. El cluster en el que se realizan las pruebas tiene capacidad de ejecutar 20 procesos en paralelo, por tanto los demas procesos estaran en espera hasta que los primeros terminen. Para la ejecucion del codigo en CUDA se invocaron 12288 bloques cada uno de ellos con 400 hilos, es decir, un numero de hilos igual a 4'915'200. Un hilo por cada elemento de la matriz.

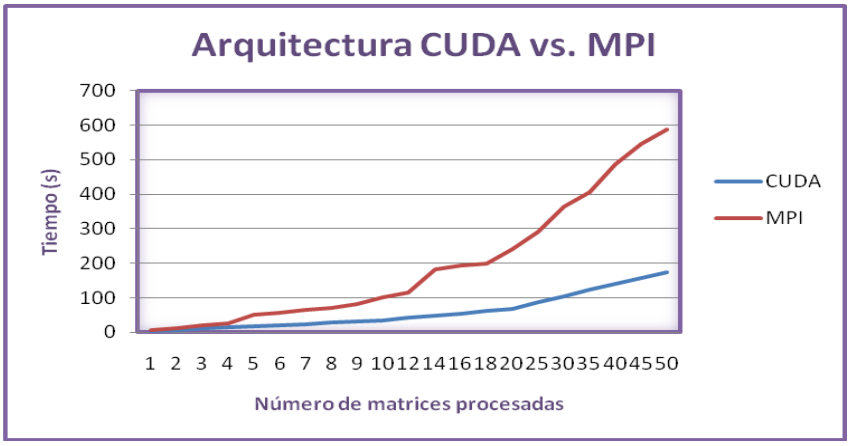


Figura 21. Tiempos de ejecución CUDA vs. MPI para un stack de 50 imágenes. Fuente: Autor

En la figura 21 se muestra el rendimiento del algoritmo en términos de tiempo. El comportamiento de la grafica para MPI es debido a que hay procesos que deben ser encolados, lo que representa un aumento en el tiempo de procesamiento. En la medida en que a cada proceso le es asignada más de una matriz, se nota un cambio drástico en el tiempo. Nótese la diferencia en un factor de 4 para el número máximo de matrices.

PRUEBA Nº4: El objetivo de esta prueba es verificar si para el cálculo de la matriz de topografía se mantiene la tendencia obtenida en la prueba Nº2. Para lograrlo en este caso se realiza pruebas sobre las matrices de varianza de tamaño 1920filas*2560columas para un stack de 140 imágenes.

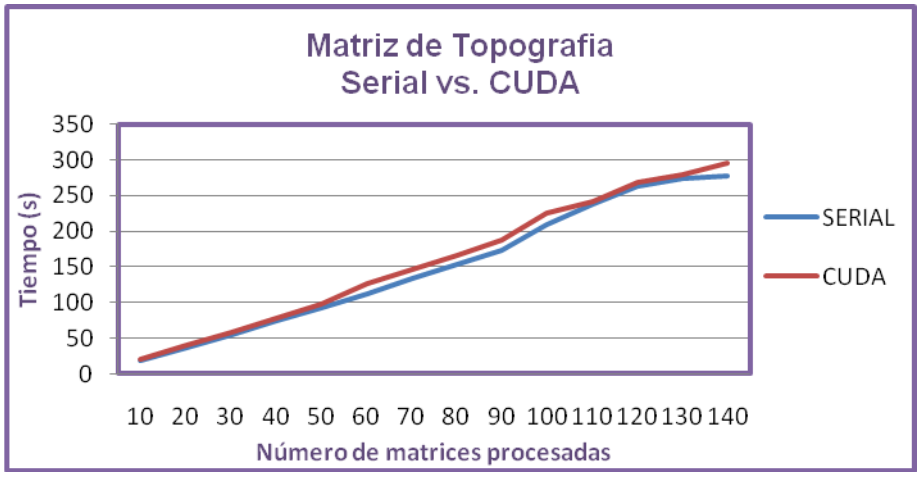


Figura 22. Calculo de la matriz de topografía a partir de un stack de 140 imágenes. Fuente: Autor

En la figura 25 se muestra un crecimiento lineal para el código serial como el código en CUDA. En los resultados obtenidos en la prueba Nº2 había una diferencia de un factor de 0.5 para las diferentes cantidades de matrices a procesar. Sin embargo a partir de los resultados de esta prueba se puede concluir que para una cantidad elevada de matrices de un formato grande, los dos códigos tienden a tener un mismo rendimiento. En la figura se observa que para ciertas cantidades de matrices el tiempo empleado en las dos arquitecturas es ligeramente igual, para otras cantidades el código serial tiene un mejor rendimiento.

- *Plataforma Grid5000*

Grid5000[17] es una platatorma Grid que permite la experimentacion en ciberstructuras y ciencias computacionales. Como instrumento científico fue diseñado y propuesto por la Agencia Nacional de Investigacion Cientifica francesa en una interaccion entre diferentes sitios de investigacion en Francia. Actualmente, reune 12 sitios en Francia y congrega 3 proyectos internacionales, un europeo, un proyecto brasilero y otro japones.

[17] www.grid5000.fr

En Grid’5000 es posible encontrar diferentes tipos de arquitecturas, como cualquier Grid de produccion. Entre las arquitecturas que encontramos, se encuentan sistemas GPGPU’s que fueron usadas para las pruebas en este proyecto, gracias a la colaboracion LIG-UIS y Universidad de Grenoble – UIS, en el marco de proyectos conjuntos.

PRUEBA Nº5: En las pruebas anteriores se ha podido comprobar la eficiencia del algoritmo en la arquitectura CUDA para el cálculo masivo de datos. Para comprobar la escalabilidad del algoritmo, se emplearon diferentes versiones de tarjetas gráficas y se analizó el rendimiento en términos de tiempo de procesamiento.

Para esta prueba se emplearon 3 tipos de tecnologías:

- Una tarjeta Quadro FX 570 empleada en las pruebas anteriores
- Una tarjeta Tesla T-10 instalada en un nodo del cluster adonis de Grid5000 con procesador Intel Xeon E5520 de 2.27 GHz con memoria de 24GB.
- Una tarjeta Tesla C2050/C2070 instalada en un equipo con procesador Intel Xeon X5650 de 2.67GHz con memoria de 74GB.

Los cálculos se realizaron sobre un stack de 90 imágenes con resolución de 1920*2560 pixeles.

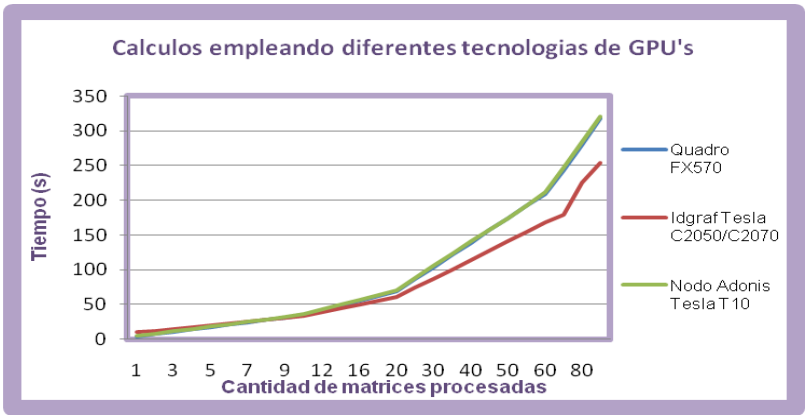


Figura 23. Tiempos de ejecución CUDA vs. MPI para un stack de 90 imágenes. Fuente: Autor

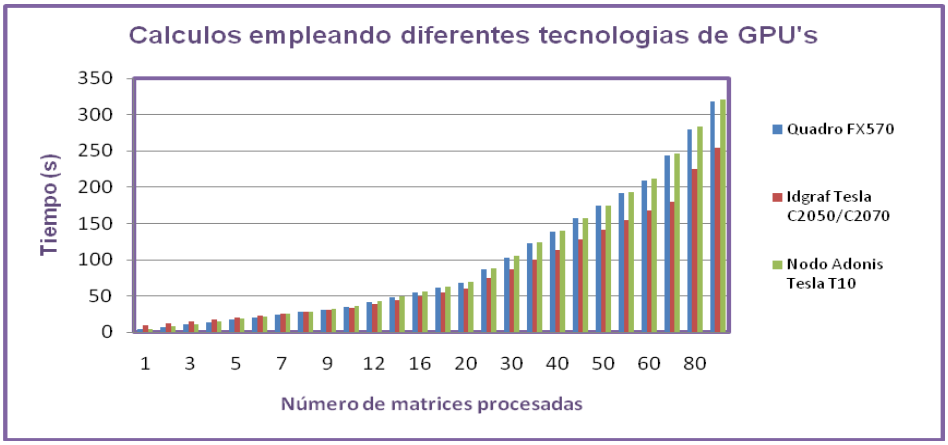


Figura 24. Gráfico de barras para los tiempos de ejecución CUDA vs. MPI. Fuente: Autor

En las figuras 22 y 23 se muestra una comparación para los tiempos de ejecución del código de cálculo de varianzas en 3 tipos de tecnologías. Como se puede observar al incrementar el número de matrices existe un mejor rendimiento para la tarjeta de gama alta Tesla C2050/C2070, en comparación con las otras tecnologías. Nótese que las tarjetas Quadro FX570 y Tesla T10 tienen tiempos similares. El rendimiento que se observa para la GPU Tesla C2050/C2070 se debe a que las versiones más recientes de GPU's tienen la capacidad de soportar una cantidad mayor de operaciones por segundo que las versiones anteriores. Sin embargo, si se analiza la figura 24, de observa que para una cantidad de matrices inferior a 10, la tarjeta de gama baja Quadro FX 570 tiene un mejor rendimiento.

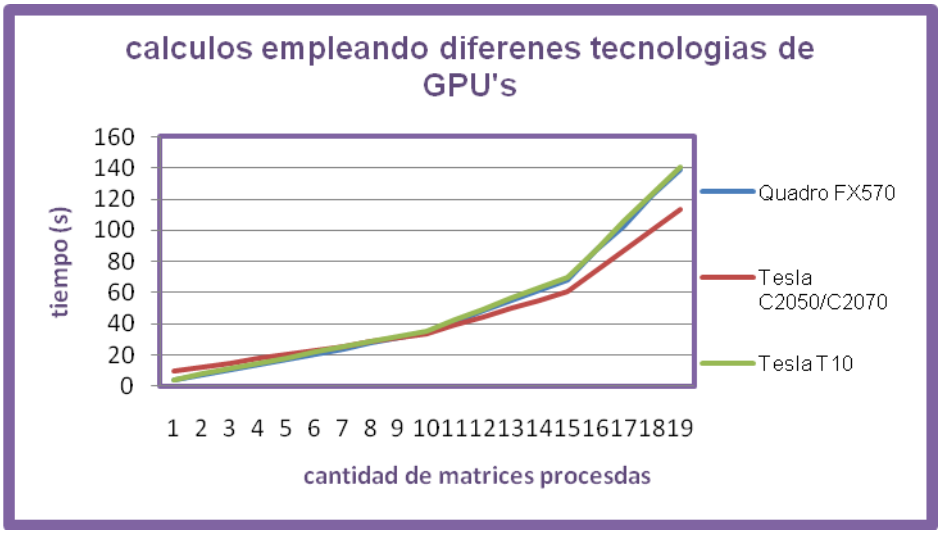


Figura 25. Tiempos de ejecución CUDA vs. MPI para las primera 20 imágenes. Fuente: Autor

PRUEBA Nº 6: Esta prueba se realizó con el fin de obtener el tiempo de cómputo del algoritmo completo para diferentes stack de imágenes. De los resultados obtenidos en las pruebas anteriores se decidió emplear CUDA® para calcular las matrices de varianzas, C para calcular la matriz de topografía y CUDA para calcular las matrices con las componentes RGB de la imagen focalizada, pues con estos códigos se obtuvieron los mejores tiempos de procesamiento.

Para no tener que ejecutar los códigos uno a la vez se creó un script en bash que ejecuta cada código una vez se obtengan los datos de entrada del mismo, es decir los resultados del código anterior (Anexo I).

Los resultados obtenidos para un stack de 93 imágenes se muestran en la figura 26.

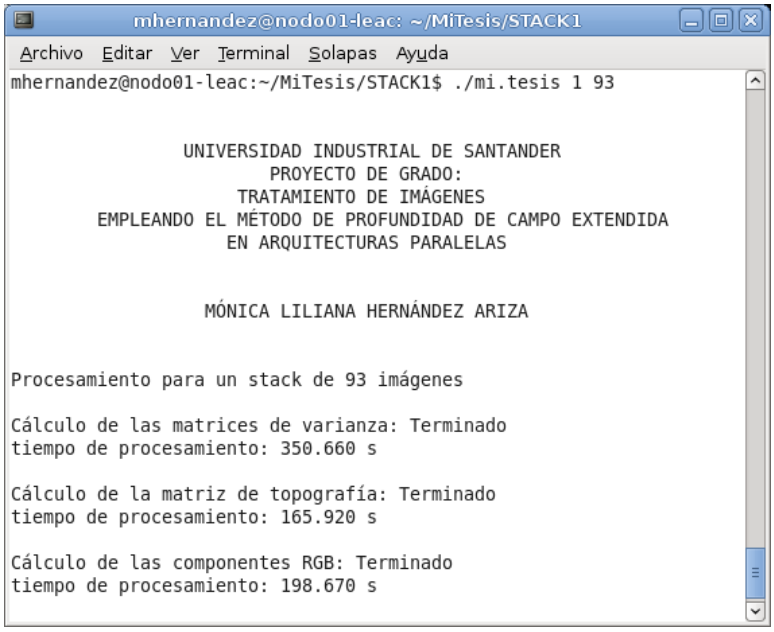


Figura 26. Tiempos de procesamiento para un stack de 93 imágenes. Fuente: Autor

Los resultados obtenidos para un stack de 149 imágenes se muestran en la figura 27.

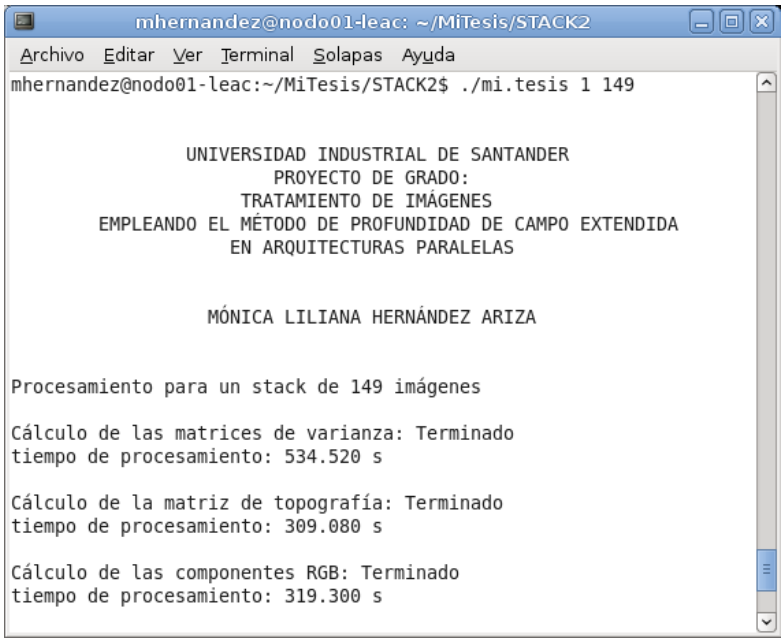


Figura 27. Tiempos de procesamiento para un stack de 149 imágenes. Fuente: Autor

PRUEBA Nº7: A partir de las matrices de varianza se obtiene una matriz de topografía. Al graficar la matriz de topografía se obtiene una imagen de relieve que permite tener una idea del volumen del objeto analizado. En esta prueba se obtuvieron 2 imágenes de topografía a partir de un stack de 93 imágenes y un stack de 149 imágenes empleando el algoritmo planteado en este proyecto de grado.

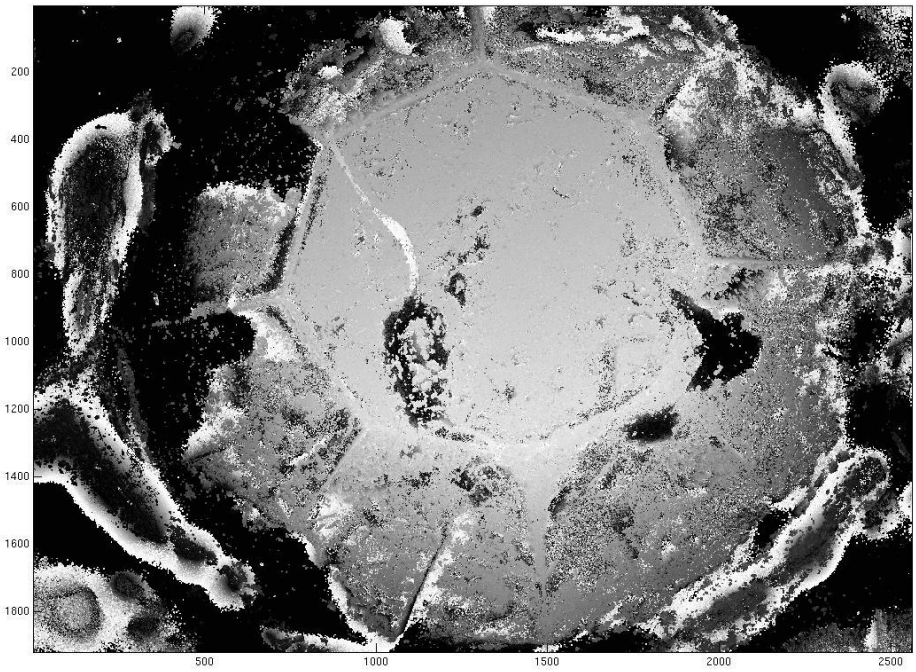


Figura 28. Imagen de topografía a partir de un stack de 93 imágenes. Fuente: Autor

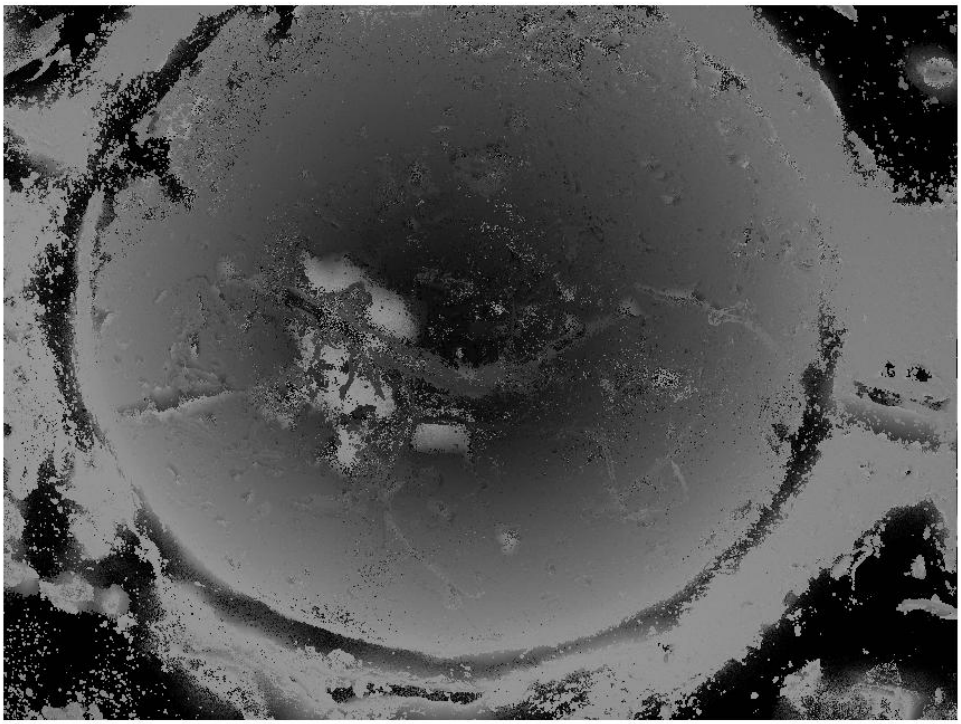


Figura 29. Imagen de topografía a partir de un stack de 149 imágenes. Fuente: Autor

PRUEBA Nº8: Una vez obtenida la matriz de topografía, se ejecuta la tercera parte del algoritmo planteado en la que se calculan las componentes RGB de la imagen final, es decir, la imagen focalizada. En esta parte, se obtiene a partir del stack los valores máximos para cada componente y se genera una imagen a partir de ellos. Para general la imagen se empleo un código en Matlab (Anexo G).

Para esta prueba se tuvieron en cuenta un stack de 93 imágenes y uno de 149 imágenes.

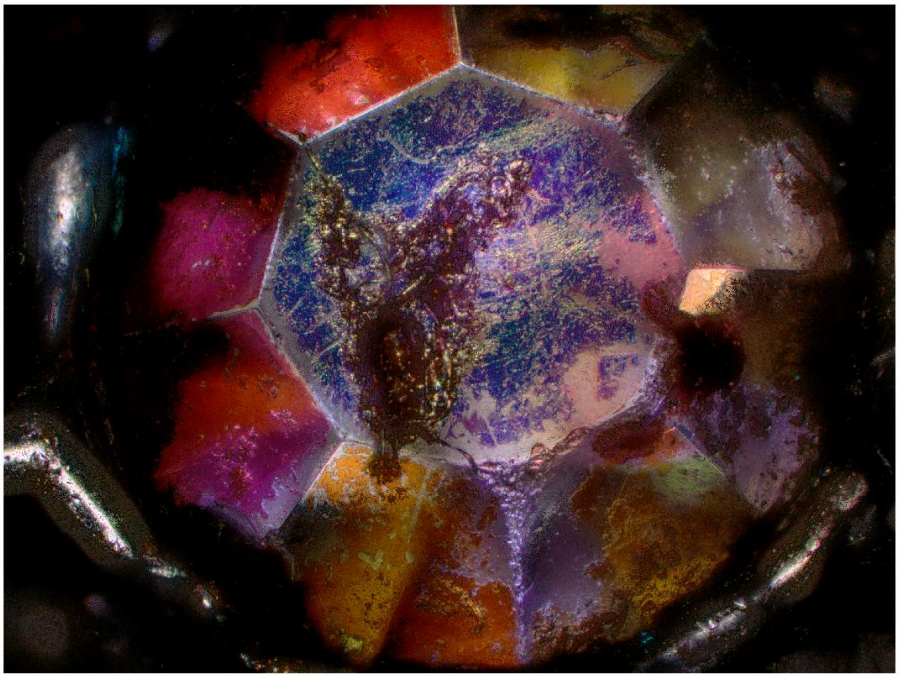


Figura 30. Imagen focalizada a partir de un stack de 93 imágenes. Fuente: Autor

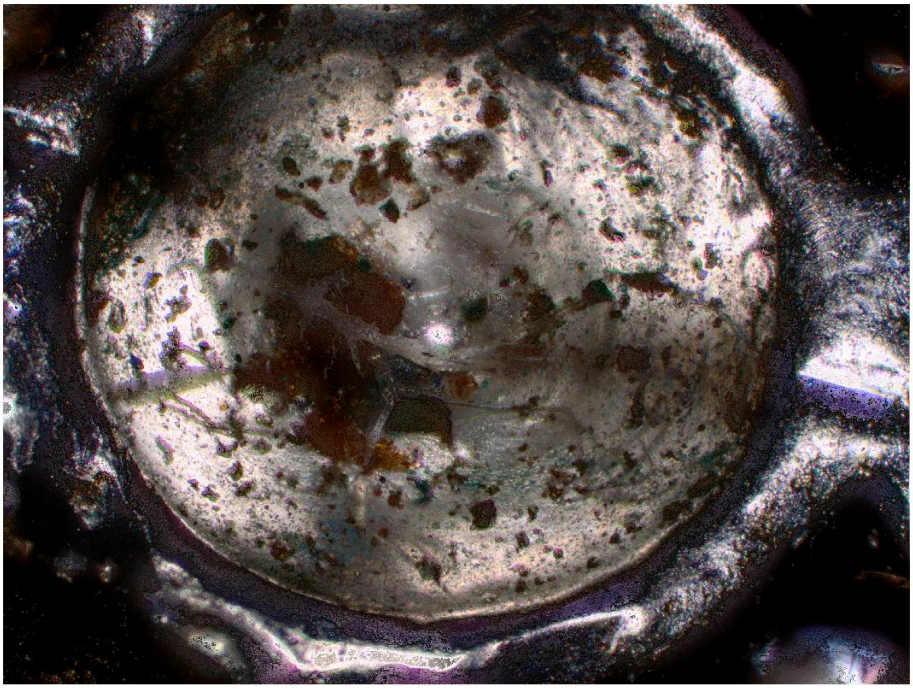


Figura 31. Imagen focalizada a partir de un stack de 149 imágenes. Fuente: Autor

En las figura 30 y 31 se muestra una imagen focalizada obtenida a partir del algoritmo planteado en este proyecto de grado. La imagen se obtuvo a partir de un stack de 93 y 149 imágenes respectivamente. Las imágenes corresponden a una piedra de colores de un accesorio y al espacio donde esta incrustada la piedra en el accesorio.

En las figuras 32-35 se muestran las imágenes renderizadas a partir de los stacks adquiridos, sin y con textura respecticamente.

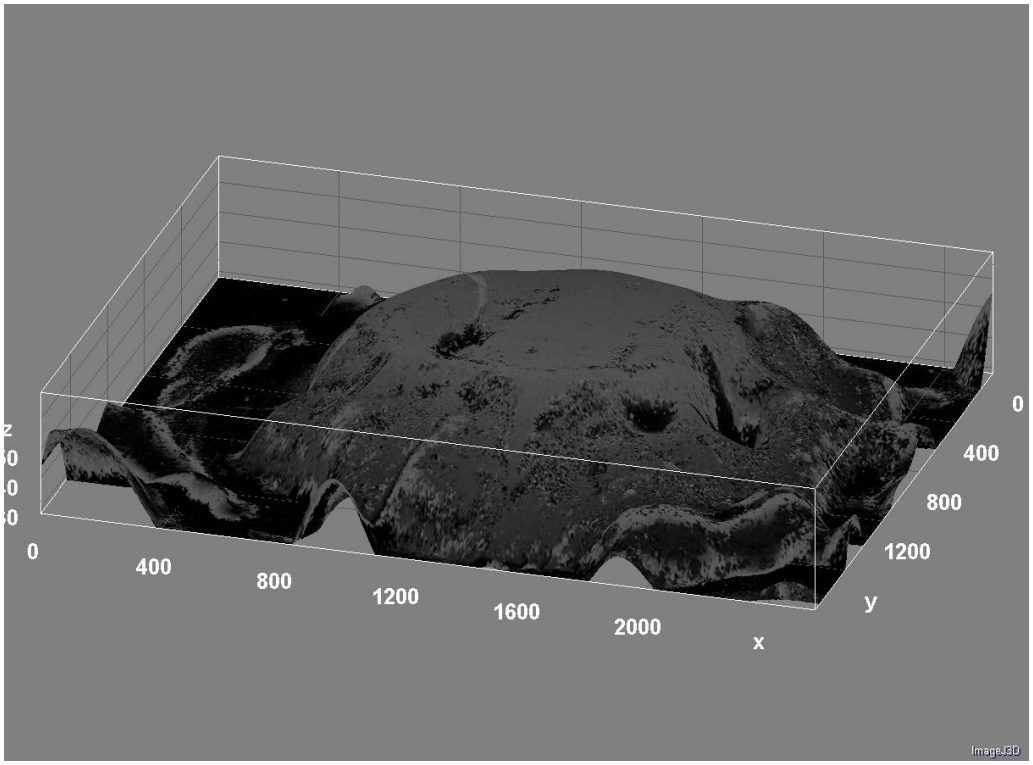


Figura 32. Render imagen de topografía stack 1 Fuente: Autor

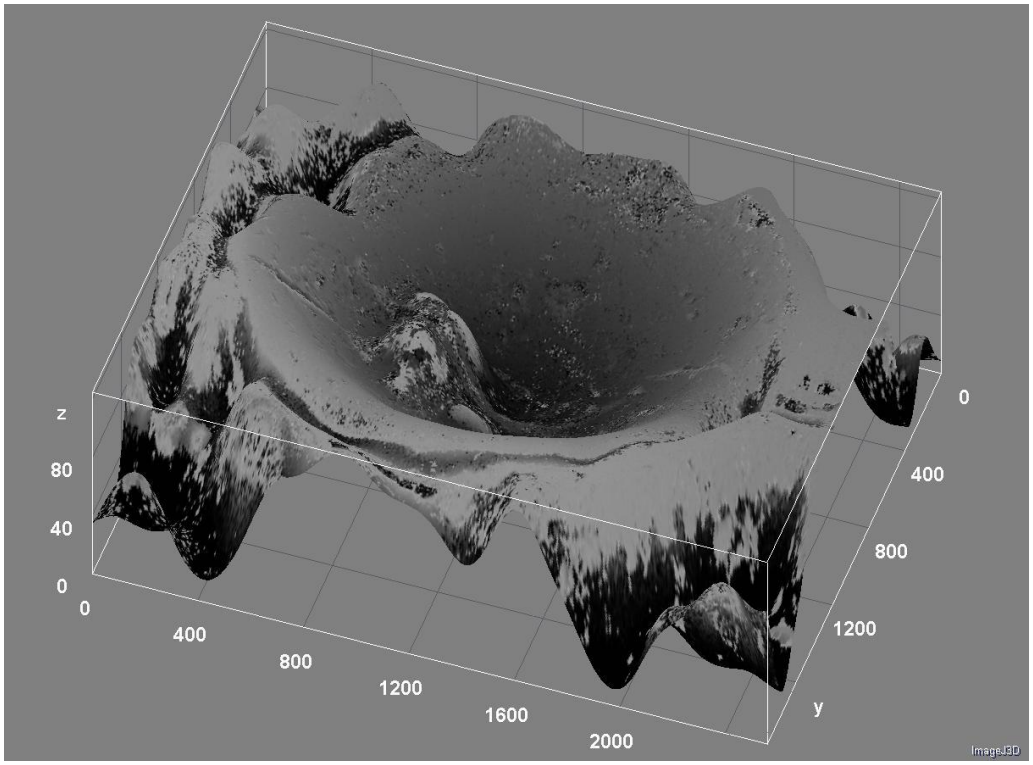


Figura 33. Render imagen de topografía stack 2 Fuente: Autor

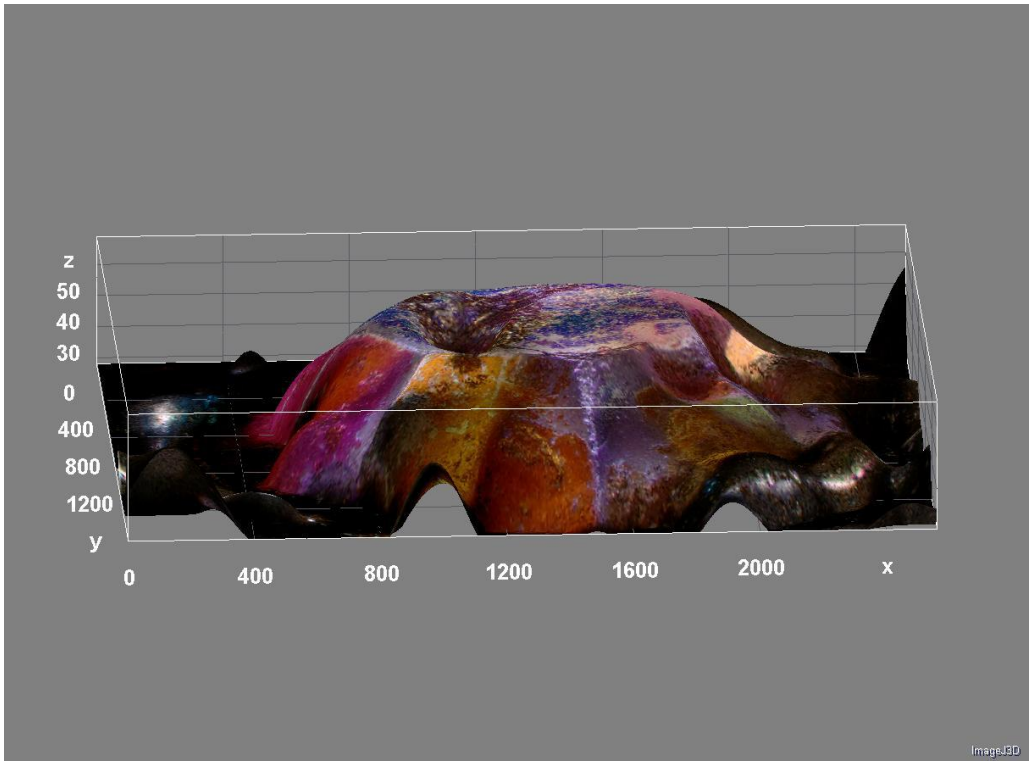


Figura 34. Render stack 1 con textura Fuente: Autor

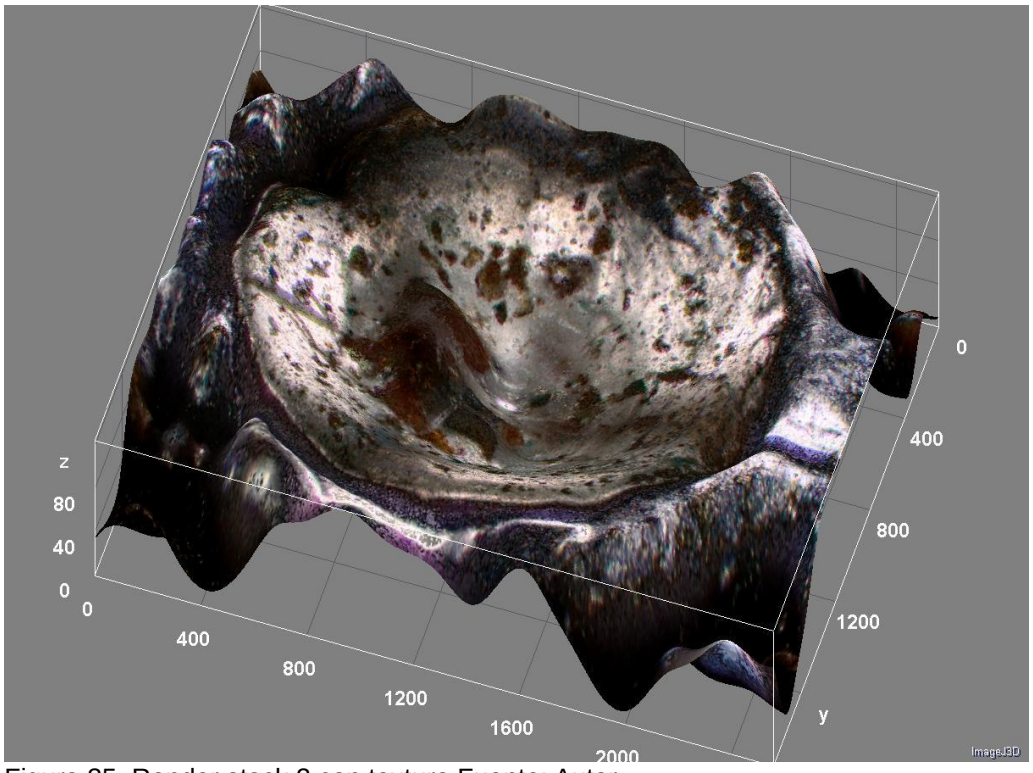


Figura 35. Render stack 2 con textura Fuente: Autor

LIMITACIONES DEL PROYECTO

Las principales limitaciones que se presentaron en el desarrollo de este proyecto estaban ligadas a las características propias de las arquitecturas empleadas. Las pruebas del algoritmo en MPI se realizaron en un cluster que contaba con 5 nodos de trabajo, lo cual limitaba la ejecución del código a 20 procesos en paralelo de manera que se tenían que encolar algunos de los procesos para poder satisfacer las necesidades del mismo código afectando el rendimiento del algoritmo.

Por otra parte, cada tarjeta GPU cuenta con una capacidad de hilos por bloque propia de cada versión. De manera que para implementar el algoritmo en las 3 tecnologías de tarjetas GPU mencionadas en las pruebas se analizaron las capacidades individuales de cada una de ellas para no afectar la escalabilidad del código.

RECOMENDACIONES

Para la realización de trabajos futuros se recomienda la creación e implementación de un servicio web que permita al usuario obtener una imagen focalizada y una imagen topografía a partir de un stack de imágenes. Donde el usuario pueda escoger diferentes opciones como el tamaño de la máscara, la arquitectura que quiere emplear ya sea cluster o GPU. En caso de ser cluster la cantidad de nodos a utilizar o en el caso de una GPU, la versión de GPU dependiendo la cantidad de datos a procesar.

Para el cálculo de las matrices de varianza se recomienda hacer pruebas empleando diferentes tamaños para la máscara. Esto permitirá realizar un análisis a los resultados obtenidos y concluir si un tamaño mayor para la máscara permite detectar de mejor manera los planos focalizados es una imagen.

Para las pruebas del algoritmo en MPI se recomienda emplear un cluster que cuente con un número mayor de nodos para analizar el rendimiento en términos de tiempo de ejecución del algoritmo al incrementar la cantidad de procesos empleados en la ejecución del algoritmo.

Otra recomendación interesante es la exploracion de la implementacion del algoritmo usando otro estandar, como podria serlo OpenCL y confrontrar el rendimiento con tarjetas similares AMD Radeon.

CONCLUSIONES

En proyectos e investigaciones previas a la realización de este trabajo se han hecho diferentes aportes y se han planteado diversas soluciones al tratamiento de imágenes. Sin embargo no se han empleado arquitecturas de alto rendimiento computacional para hacer tratamiento de imágenes utilizando el método de profundidad de campo extendida. Con la realización de este proyecto se logró proponer una nueva solución al tratamiento de imágenes utilizando EDF, una manera de procesar imágenes que no había sido planteada en investigaciones previas.

Gran parte de los problemas que se pretenden resolver en el campo de la investigación requieren una capacidad de cómputo elevada que no puede ser soportada por un equipo que cuenta con un hardware de características convencionales. Para este tipo de problemas, emplear arquitecturas de alto rendimiento computacional como clusters y tarjetas graficas se convierte en una alternativa que permite acelerar el proceso de obtención de resultados. En algunos casos donde no se ha llegado a un resultado, una arquitectura HPC[18] permite obtener los resultados que se buscan mediante la optimización de soluciones propuestas o el planteamiento de nuevas soluciones.

Los paradigmas de programación paralela como MPI y CUDA son herramientas que contribuyen a que el diseño y la programación de algoritmos empleen los recursos de una arquitectura en busca de un aprovechamiento máximo. Estos paradigmas permiten dividir un problema en tareas más sencillas para ser tratadas por diferentes recursos. Sin embargo, el proceso de comunicación entre estos recursos puede afectar el rendimiento de un determinado algoritmo

[18] Computación de alto rendimiento

debido a que es necesario enviar paquetes entre los diferentes recursos de la arquitectura.

Cuando se emplea una arquitectura HPC para tratar un problema en paralelo, es importante identificar que partes del algoritmo que se diseña pueden ser tratadas en paralelo, cuales no pueden serlo y en cuales partes no es conveniente emplearlas. La programación paralela no implica que un algoritmo sea completamente paralelo, pues hay instrucciones que deben ser procesadas de manera serial para obtener mejores resultados.

A partir del método de profundidad de campo extendida se obtienen imágenes con un porcentaje de nitidez satisfactorio. Al emplear métodos artificiales que permiten hacer empalmes a partir de un stack de imágenes es posible destacar los datos contenidos en cada imagen de tal manera que se obtenga una imagen que contenga toda la información importante de las imágenes individuales.

A partir de las pruebas realizadas se comprobó que para realizar operaciones propias del tratamiento de imágenes, la arquitectura CUDA® es la mejor opción en términos de tiempo de ejecución, pues ofrece un mejor rendimiento en cuanto a tiempo de procesamiento para un número determinado de matrices, debido a la granularidad fina del problema. Aunque al emplear un cluster se está dividiendo el problema en tareas más sencillas lo cual permite obtener un mejor rendimiento que el que se obtiene en algoritmos seriales, este rendimiento se ve afectado por los costos en comunicación y la latencia que se presenta cuando hay envío de paquetes a través de la red.

Al realizar pruebas sobre diferentes tecnologías de tarjetas GPU’s se comprobó que las tarjetas de gama alta tienen un buen desempeño para hacer procesamiento masivo de datos. Sin embargo, cuando se realiza procesamiento sobre un volumen de datos pequeño, en las tarjetas de gama baja se obtiene un mejor rendimiento en términos de tiempo. De lo anterior se concluye que no en todos los casos la mejor alternativa es la que cuenta con un mayor número de recursos. Al igual que para determinar en que casos se debe emplear un cluster y en cuales una GPU, es importante identificar en que situaciones es conveniente emplear una GPU de gama alta y en cuales una GPU de gama baja.

Mediante la obtención de una imagen de topografía y una imagen focalizada de un objeto real a partir de un stack de imágenes es posible reconstruir una imagen que proporcione una sensación de tridimensionalidad. Este tipo de imágenes permite tener mundos virtuales más acordes con la realidad que se está analizando, donde estos mundos virtuales destacan los detalles más finos de lo que se observa. En algunos casos, como los mostrados en las pruebas que se realizaron en este proyecto, la realidad que se estudia no es completamente clara para el observador, pues son objetos u organismos que carecen de formas definidas o que sus detalles no pueden ser percibidos por el ojo humano.

BIBLIOGRAFIA

[1] Wittwer Tobias, an Introduction to Parallel Programming, First edition 2006, VSSD.

[2] Karniadakis George Em, Kirby II Robert M., Parallel Scientific Computing in C++ and MPI, A seamless approach to parallel algorithms and their implementation, Cambridge University Press.

[3] Snir Marc , Otto Steve , Huss-Lederman Steven , Walker David, and Dongarra Jack , MPI: The Complete Reference, The MPI Press, Cambridge, Massachusetts, London, England.

[4] Garland Michael, Kirk David B., Understanding Throughput-Oriented Architecture, Communications of the ACM, November 2010.

[5] NVIDIA, GPU Computing, http://www.nvidia.es.

[6] Dongarra Jack et all, Sourcebook of parallel Computing**,** Morgan Kaufmann Publishers, 2003 by Elsevier Science (USA).

[7] Nvidia, NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, Versión 1.1, 2007.

[8] Sanders Jason, Kandrot Edward, CUDA by Example: An Introduction to General-Purpose GPU Programming, First printing, July 2010, Addison-Wesley.

[9]Gonzales Rafael C., Woods Richard E, Tratamiento digital de imagenes, Addison-Wesley, 1996.

[10] Ben-Eliezer Eyal, Zalevsky Zeev, Marom Emanuel, Konforti Naim, All-Optical Extended Depth of Field, IOP Publishing Ltd 2003.

[11]Ecole Polytechnique Federale De Lausanne, Biomedical Imaging Group, Extended Depth of Field, http://bigwww.epfl.ch/demo/edf/

[12] Zhiyi Yang, Yating Zhu, Yong Pu, Parallel Image Processing Based on CUDA, CSSE '08 Proceedings of the 2008 International Conference on Computer Science and Software

Engineering - Volume 03

[13] In Kyu Park , Incheon Nitin Singhal , Suwon Man Hee Lee , Incheon Sungdae Cho , Suwon Chris Kim, Design and Performance Evaluation of Image Processing Algorithms on GPUs, IEEE Transactions on Parallel and Distributed Systems, Volume 22 Issue 1, January 2011

[14] Burkhardt Hans, Lang Bernhard, Nolle Michael, Aspects of Parallel Image Processing Algorithms and Structures, Technical Report Aspects of Parallel Image Processing Algorithms and Structures Universitaet Hamburg Hamburg, Germany, Germany ©1990

[15] Atanaska Bosakova-Ardenska, Simeon Petrov, Naiden Vasilev, Implementation of parallel algorithm "conveyer processing" for images processing by filter 'mean', CompSysTech '07 Proceedings of the 2007 international conference on Computer systems and technologies ACM New York, NY, USA ©2007

[16] Seinstra F. J., Koelma D., Geusebroek J. M., A software architecture for user transparent parallel image processing, Parallel Computing - Parallel computing in image and video processing Volume 28 Issue 7-8, August 2002

[17] Tu Xionggang; Chen Jun, Parallel image processing with OpenMP, Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on

[18] Tessens, L.; Ledda, A.; Pizurica, A.; Philips, W., Extending the Depth of Field in Microscopy Through Curvelet-Based Frequency-Adaptive Image Fusion, Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on

[19] Bradley, A.; Wildermoth, M.; Mills, P.; Virtual Microscopy with Extended Depth of Field, This paper appears in: Digital Image Computing: Technqiues and Applications, 2005. DICTA ' 05. Proceedings

[20] Wang Jin-jiang; Xu Ming; Liu Wen-yao; To Extend the Depth of Field Based on Image Processing, Image and Signal Processing, 2009. CISP '09. 2nd International Congress on

[21] Boddeti, V.N.; Kumar, B.V.K.V.; Extended-Depth-of-Field Iris Recognition Using Unrestored Wavefront-Coded Imagery. Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on

[22] ImageJ, http://rsbweb.nih.gov/ij/, ImageJ Image Processing and Analysis in Java.

[23] Hernández M. L., Camargo F. L., Lobo A., Barrios Hernández C. J., Plata Gómez A.' Sierra D. A., Implementación del Método de Profundidad de Campo Extendida en Arquitecturas

Paralelas, Conferencia Latinoamericana de Computación de Alto Rendimiento CLCAR 2011 Colima, México. http://clcar.itcolima.edu.mx/es

ANEXOS

ANEXO A: Código en matlab que obtiene las componentes RGB para casa imagen del stack.

Código en matlab que obtiene las componentes RGB para casa imagen del stack. Cada componente es almacenada en un archivo en forma de matriz. Por cada imagen se crea una carpeta donde se almacenan las 3 matrices obtenidas a partir de cada componente. Para hacer más fácil la lectura de las imágenes del stack, se creó un script en bash que lee cada imagen y cambia su nombre por la posición que ocupa en el stack dicha imagen y un script que crea una carpeta por cada imagen.

Script para cambiar el nombre de cada imagen:

```
#!/bin/bash
for ((i=$1;i<=$2;i+=1)); do
imagen=`ls | grep Image | sort -r | tail -n 1`
mv $imagen $i
done
```

Script para crear una carpeta por cada imagen:

```
#!/bin/bash
for ((i=$1;i<=$2;i+=1)); do
mkdir -p $i
done
```

Código en matlab

```
format short;
for m=1:93
   S=sprintf('stack/%d',m);
I=imread(S);
I2=double(I);

R=I2(:,:,1);
G=I2(:,:,2);
B=I2(:,:,3);

Rf=sprintf('RGB/%d/R',m);
Gf=sprintf('RGB/%d/G',m);
Bf=sprintf('RGB/%d/B',m);
```

```
fid1 = fopen(Rf,'w');
fid2 = fopen(Gf,'w');
fid3 = fopen(Bf,'w');

for i=1:1920
    for j=1:2560
        fprintf(fid1,'%d ',R(i,j));
        fprintf(fid2,'%d ',G(i,j));
        fprintf(fid3,'%d ',B(i,j));
    end
    fprintf(fid1,'\n');
    fprintf(fid2,'\n');
    fprintf(fid3,'\n');
end

fclose(fid1);
fclose(fid2);
fclose(fid3);
end
```

ANEXO B: Código en MPI para el cálculo de las matrices de varianza

Código en C que incluye librerías de MPI para calcular las matrices de varianzas de cada imagen a partir de los equivalentes numéricos de la componente verde para cada pixel.

Código MPI

```
#include “mpi.h”
#include <stdio.h>
#include <stdlib.h>
#include<time.h>

int id,size,dest,nodes;
int dimx=1920, dimy=2560;

int main (int argc, char *argv[]){

        //********declaración de variables********

        MPI_Request request;
        MPI_Status status;
        int t,init,fin;

        init=atoi(argv[1]);
        fin=atoi(argv[2]);

        //********inicialización MPI*********

        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&id);
        MPI_Comm_size(MPI_COMM_WORLD,&size);

        nodes=size-1;

        printf(“Yo soy %d de %d\n”,id,size);
        //***********MAESTRO************

        if (id == 0){

        //***********DeclaraciÃ³n de variables en el maestro***************

                int A[dimx][dimy];
                double Var[dimx][dimy];
                int c,r,k,s,d,f,h,g,workers,imag,imag_n;
```

```
        float t;
        clock_t tinicio, tfinal;

//*************Lectura matriz de datos*****************

        tinicio=clock();
        imag=0;
        FILE* 85unctio;

                for(dest=1;dest<=nodes;dest++){
                        imag++;
                        char rutaG[]=”RGB/”;
                        char matriz[]=”/G”;
                        85uncti(rutaG,”%s%d%s”, rutaG,imag,matriz);

                        85unctio=fopen(rutaG,”r+”);
                                for(r=0;r<dimx;r++){
                                        for(c=0;c<dimy;c++){
                                                fscanf(archivo,”%d”,&A[r][c]);

                                        }
                                }
                        fclose(archivo);

//************envio matriz A a cada nodo************


        MPI_Send(&A, dimx*dimy, MPI_INT, dest,
99,MPI_COMM_WORLD);
        MPI_Send(&imag, 1, MPI_INT, dest, 99,MPI_COMM_WORLD);

                }

//************recepción de matrices de varianza calculada en los
nodos*************

                for(dest=1;dest<=nodes;dest++){
                        for(r=0;r<dimx;r++)
                        for(c=0;c<dimy;c++)
                                Var[r][c]=0;

MPI_Recv(&Var, dimx*dimy, MPI_DOUBLE, dest, 99, MPI_COMM_WORLD
,&status);
MPI_Recv(&imag_n, 1, MPI_INT, dest, 99, MPI_COMM_WORLD,&status);

                        FILE* resultados;
```

```
                                char rutaV[]="VARIANZAS/";
                                86uncti(rutaV,"%s%d",rutaV,imag_n);
                                resultados=fopen(rutaV,"w+");

                                for(k=0;k<dimx;k++){
                                        fprintf(resultados,"\n");
                                for(s=0;s<dimy;s++){
                                                fprintf(resultados,"%7.2f ",Var[k][s]);
                                          }
                                 }
                        }

                tfinal=clock();
                t = ((float)tfinal-(float)tinicio)/CLOCKS_PER_SEC;
                printf("tiempo de ejecuciÃ³n para %d matrices: %6.5f\n",fin-
init+1,t);
        }

        //************NODOS************

        else
        {

        //**********Declaración de variables en los nodos************

                int l,p,I,j,imag_i;
                int A_n[dimx][dimy],B[dimx+2][dimy+2];
                double Var_n[dimx][dimy];
                for(l=0;l<dimx;l++)
                        for(p=0;p<dimy;p++)
                                Var_n[l][p]=0;

        //************recepción matriz de datos***********


MPI_Recv(&A_n,dimx*dimy,MPI_INT,0,99,MPI_COMM_WORLD,&status);
        MPI_Recv(&imag_i, 1, MPI_INT, 0, 99, MPI_COMM_WORLD,&status);


        //***********incrementar filas y columnas de la matriz de datos*******

        for(i=0;i<dimx+2;i++)
                for(j=0;j<dimy+2;j++)
                        B[i][j]=((i==0 || j==0 || i==dimx+1 || j==dimy+1) ? 0:A_n[i-1][j-
1]);
```

```
        //***********calcular la matriz de varianza*************

                int M[9];
                int X,m,n;
                double Xprom,Y;

                for(i=1;i<=dimx;i++){
                        for(j=1;j<=dimy;j++){

                                Xprom=0.f,Y=0.f, X=0;

                                M[0]=B[i-1][j-1];
                                M[1]=B[i-1][j];
                                M[2]=B[i-1][j+1];
                                M[3]=B[i][j-1];
                                M[4]=B[i][j];
                                M[5]=B[i][j+1];
                                M[6]=B[i+1][j-1];
                                M[7]=B[i+1][j];
                                M[8]=B[i+1][j+1];

                                for(m=0;m<9;m++)
                                        X+=M[m];
                                Xprom=((double)X)/9;


                                for(n=0;n<9;n++)
                                        Y+=(Xprom-M[n])*(Xprom-M[n]);
                                Var_n[i-1][j-1]=Y/9;
                        }
                }

        //*********envio de la matriz de varianza al maestro***********

        MPI_Send(Var_n, dimx*dimy, MPI_DOUBLE, 0, 99,
MPI_COMM_WORLD);
        MPI_Send(&imag_i, 1, MPI_INT, 0, 99,MPI_COMM_WORLD);

        }

        //*********finalizar MPI*************
  MPI_Finalize();

}
```

ANEXO C: Código en CUDA® para el cálculo de las matrices de varianza

Código en C que incluye librerías de CUDA® para calcular las matrices de varianzas de cada imagen a partir de los equivalentes numéricos de la componente verde para cada pixel.

Código CUDA

```
#include<stdio.h>
#include<stdlib.h>
#include<cuda.h>
#include<time.h>

//************variables globales***************

int msk=3, dimx=1920,dimy=2560,tam_imag=1920*2560;

//******************kernel*******************

__global__ void kernel (int *B_d,float *var_d){

        int idx = threadIdx.x + blockIdx.x*blockDim.x;
        88unctio = threadIdx.y + blockIdx.y*blockDim.y;
        int offset=idx + idy*blockDim.x*gridDim.x;

        int id=offset;
        int I;
        float X=0.f,Xprom=0.f,Y=0.f;
        int dimy=2560,tam_imag=1920*2560,msk=3;
        var_d[id]=0;

                if(offset<tam_imag){
                        int dimy_B=dimy+2;

                        offset+=2*idy;
                        int id_p=offset+(dimy+msk);

                        int M_d[9];

                        M_d[0]=B_d[offset];
                        M_d[1]=B_d[offset+1];
                        M_d[2]=B_d[offset+2];
                        M_d[3]=B_d[id_p-1];
                        M_d[4]=B_d[id_p];
                        M_d[5]=B_d[id_p+1];
```

```
                        M_d[6]=B_d[(id_p-1)+dimy_B];
                        M_d[7]=B_d[id_p+dimy_B];
                        M_d[8]=B_d[(id_p+1)+dimy_B];

                        for(i=0;i<msk*msk;i++)
                                X+=M_d[i];
                        Xprom=((float)X)/(msk*msk);

                        for(i=0;i<msk*msk;i++)
                                Y+=(Xprom-M_d[i])*(Xprom-M_d[i]);
                        var_d[id]=Y/(msk*msk);
                }
}

//*****************89unction main*********************

        int main(int argc,char* argv[]){


        //***************declaraciÃ³n de variables**************

                int I,j,m,cont,tam_B, init,fin;
                init=atoi(argv[1]);
                fin=atoi(argv[2]);

                tam_B=(dimx+2)*(dimy+2);

                FILE *arch, *matrizG;

                int **A;
                int B[dimx+2][dimy+2];

                float t;
                clock_t tinicio, t_GPU;
                tinicio=clock();

                int *B_d, *B_h;
                float *var_d,*var_h;

                        for(int d=init;d<=fin;d++){

                        //******************declaracion de variables***************

                                B_h=(int *)malloc(sizeof(int)*tam_B);
                                cudaMalloc((void**)&B_d, tam_B*sizeof(int));
```

```
                                var_h=(float *)malloc(sizeof(float)*tam_imag);
                                cudaMalloc((void**)&var_d,tam_imag*sizeof(float));

                                A=(int **)malloc(sizeof(int)*dimx);
                                for(i=0;i<dimx;i++)
                                A[i]=(int*)malloc(sizeof(int)*dimy);


        //*****************calculo matriz B****************

        char ruta1[]="MiTesis/";
        90uncti(ruta1, "%s%d%s","RGB/",d,"/G");
         matrizG=fopen(ruta1,"r+");

        for(i=0;i<dimx;i++)
                for(j=0;j<dimy;j++)
                        fscanf(matrizG, "%d", &A[i][j]);
                fclose(matrizG);

        cont=0;
        for(i=0;i<dimx+2;i++){
                for(j=0;j<dimy+2;j++){
                        B[i][j]=((i==0 || j==0 || i==dimx+1 || j==dimy+1) ? 0:A[i-1][j-
1]);
                        B_h[cont]=B[i][j];
                        cont++;
                }
        }

//******************llamado de kernel*******************

        dim3 Grid(128,96);
        dim3 Block(20,20);

cudaMemcpy(B_d,B_h,sizeof(int)*tam_B,cudaMemcpyHostToDevice);

        kernel<<<Grid,Block>>>(B_d,var_d);

        cudaMemcpy(var_h,var_d,sizeof(float)*tam_imag,cudaMemcpyDeviceTo
Host);


//****************almacenamiento matriz de varianza**************

        char rutaV[]="VARIANZAS/";
        90uncti(rutaV, "%s%d", rutaV,d);
```

```
        arch=fopen(rutaV,"w+");

        for(m=0;m<tam_imag;m++){
                if(m%dimy==0 && m!=0){
                fprintf(arch,"\n");
                }
                fprintf(arch,"%f ",var_h[m]);
        }
        fclose(arch);
free(B_h);
free(var_h);
free(A);
cudaFree(var_d);
cudaFree(B_d);
}


t_GPU=clock();
t = ((float)t_GPU-(float)tinicio)/CLOCKS_PER_SEC;
printf("tiempo de procesamiento para calcular varianzas de %d matrices: %6.3f
s\n",fin-init+1,t);

return 0;

}//FIN 91unction main()
```

ANEXO D: Código en C para calcular la matriz de topografía.

Código en C para calcular la matriz de topografía.

Código C

```
//*************inclución de librerias***************

#include<stdio.h>
#include<stdlib.h>
#include<time.h>

//************variables globales***************

int N=93, dimx=1920, dimy=2560, tam_imag=1920*2560;

//****************declaración de funciones****************

	float** leerMatrizVarianza(int d);

//*****************función main**********************

	int main(int argc,char* argv[]){

		//***************declaración de variables**************

		int i,j,k,temp, **top;
		float **max,**varianza, t;
		clock_t tinicio, t_GPU;
		tinicio=clock();

		//************inicialización de variables***************
			max=(float **)malloc(sizeof(float)*dimx);
				for(i=0;i<dimx;i++)
					max[i]=(float*)malloc(sizeof(float)*dimy);

			top=(int **)malloc(sizeof(int)*dimx);
				for(i=0;i<dimx;i++)
					top[i]=(int*)malloc(sizeof(int)*dimy);

			varianza=(float **)malloc(sizeof(float)*dimx);
				for(i=0;i<dimx;i++)
					varianza[i]=(float*)malloc(sizeof(float)*dimy);


		//***************cálculo de la mayor varianza************
```

```
 temp=1;
max=leerMatrizVarianza(temp);
for(i=0;i<dimx;i++)
  for(j=0;j<dimy;j++)
        top[i][j]=temp;

    for(k=2;k<=N;k++){
        printf("k=%d\n",k);
        varianza=leerMatrizVarianza(k);

        for(i=0;i<dimx;i++){
            for(j=0;j<dimy;j++){
                if(varianza[i][j]>max[i][j]){
                    top[i][j]=k;
                    max[i][j]=varianza[i][j];
                }
            }
        }
    }
    free(varianza);

//***************Almacenamiento matriz topográfica************

    FILE *topo;
    topo=fopen("Resultados/topos","w+");
    for(i=0;i<dimx;i++){
        for(j=0;j<dimy;j++){

            fprintf(topo,"%d ",top[i][j]);
            if(j%2559==0 && j!=0)
                fprintf(topo,"\n");
        }
    }
    fclose(topo);
    free(max);
t_GPU=clock();
t = ((float)t_GPU-(float)tinicio)/CLOCKS_PER_SEC;
printf("tiempo de procesamiento: %6.3f s\n",t);

return 0;

}//FIN función main()

//*****************leerMatrizVarianza***************
float** leerMatrizVarianza(int d){
```

```
        int i,j;

        char rutavar[]="VARIANZAS/";
        sprintf(rutavar,"%s%d",rutavar,d);

        FILE* archivo;
        archivo=fopen(rutavar,"r+") ;

        float **var=(float **)malloc(sizeof(float)*dimx);
              for(i=0;i<dimx;i++)
                    var[i]=(float*)malloc(sizeof(float)*dimy);

        for(i=0;i<dimx;i++)
              for(j=0;j<dimy;j++){
                    fscanf(archivo,"%f",&var[i][j]);
                }
        fclose(archivo);

        return var;
}
```

ANEXO E: Código C que incluye librerías de CUDA para calcular la matriz de topografía.

Código C que incluye librerías de CUDA para calcular la matriz de topografía.

Código CUDA

```
//*************inclución de librerias***************

#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<cuda.h>

//************variables globales***************

int N=93, dimx=1920, dimy=2560, tam_imag=1920*2560;
//**********KERNEL**************

__global__ void kernel (float *max, float *var, int *top, int k){

        int idx=threadIdx.x + blockIdx.x*blockDim.x;
        int tam_imag=1920*2560;

        if(idx<tam_imag){

                if(var[idx]>max[idx]){
                top[idx]=k;
                max[idx]=var[idx];
                }
        }
}

float *leerMatrizVarianza(int d);

//*****************función main**********************

        int main(int argc,char* argv[]){

                //***************declaraciÃ³n de variables**************
                int i,k,temp;

                int *top_d;
                int top_h[dimx*dimy];
                cudaMalloc((void **)&top_d,sizeof(int)*dimx*dimy);
```

```
		float *max_d,  *var_d;
		float *max_h, *var_h;

		var_h=(float *)malloc(sizeof(float)*dimx*dimy);
		max_h=(float *)malloc(sizeof(float)*dimx*dimy);
		cudaMalloc((void **)&max_d,sizeof(float)*dimx*dimy);
		cudaMalloc((void **)&var_d,sizeof(float)*dimx*dimy);

		float t;
		clock_t tinicio, t_GPU;
		tinicio=clock();

		//***************cálculo de la mayor varianza************
			temp=1;
			max_h=leerMatrizVarianza(temp);
			for(i=0;i<dimx*dimy;i++)
				top_h[i]=temp;

				for(k=2;k<=N;k++){
					printf("k=%d\n", k);
					var_h=leerMatrizVarianza(k);
cudaMemcpy(max_d,max_h,sizeof(float)*dimx*dimy,cudaMemcpyHostToDevice
);
cudaMemcpy(var_d,var_h,sizeof(float)*dimx*dimy,cudaMemcpyHostToDevice);
cudaMemcpy(top_d,top_h,sizeof(int)*dimx*dimy,cudaMemcpyHostToDevice);

kernel<<<12288,400>>>(max_d,var_d,top_d,k);

cudaMemcpy(top_h,top_d,sizeof(int)*dimx*dimy,cudaMemcpyDeviceToHost);
cudaMemcpy(max_h,max_d,sizeof(float)*dimx*dimy,cudaMemcpyDeviceToHost
);
}
		cudaFree(max_d);
		cudaFree(var_d);
		cudaFree(top_d);

FILE *topo;
topo=fopen("Resultados/topo","w+");
for(i=0;i<dimx*dimy;i++){
	if(i%dimy==0 && i!=0)
		fprintf(topo,"\n");
		fprintf(topo,"%d ",top_h[i]);
}
fclose(topo);
```

```
	t_GPU=clock();
	t = ((float)t_GPU-(float)tinicio)/CLOCKS_PER_SEC;
	printf("tiempo de procesamiento: %6.3f s\n",t);

}//FIN función main()

//*****************leerMatrizVarianza***************

float* leerMatrizVarianza(int d){

	int i;
	char rutavar[]="VARIANZAS/";
	sprintf(rutavar,"%s%d",rutavar,d);

	FILE* archivo;
	archivo=fopen(rutavar,"r") ;

	float *var;
	var=(float *)malloc(sizeof(float)*dimx*dimy);

	for(i=0;i<dimx*dimy;i++)
		fscanf(archivo,"%f",&var[i]);
	fclose(archivo);
	return var;
}
```

ANEXO F: Código en CUDA para el cálculo de las componentes RGB

Código en C que incluye librerías en CUDA para calcular las componentes RGB de la imagen focalizada.

Código CUDA

```
//****librerias****

#include<stdio.h>
#include<stdlib.h>
#include<cuda.h>
#include<time.h>

//****Variables globales****

int N=93, dimx=1920,dimy=2560,tam_imag=1920*2560;

//****Kernel: Función del device****

__global__ void Kernel(int *R_d, int *G_d, int *B_d, int *T_d, int *Rf, int *Gf, int
*Bf, int d){

        int idx = threadIdx.x + blockIdx.x*blockDim.x;

        int tam_imag;
        tam_imag=1920*2560;
        if(idx<tam_imag)
                if(T_d[idx]==d){

                        Rf[idx]=R_d[idx];
                  Gf[idx]=G_d[idx];
                        Bf[idx]=B_d[idx];
                }
}

//****Función main()****

int main(int argc,char* argv[]){

//****declaración de variables para el host y device****

int i, j, d,  cont;
int *R_h, *R_d, *G_h, *G_d, *B_h, *B_d, *T_d, *T_h;
int *R, *G, *B;
int *Rf, *Gf, *Bf;
```

```
FILE *file, *Red, *Green, *Blue;
FILE *ArchivoR, *ArchivoG, *ArchivoB;
//****Leer archivo de la matriz topografica****

file=fopen("Resultados/topo","r+");

//******matriz R host y device********

R_h=(int *)malloc(sizeof(int)*tam_imag);
cudaMalloc((void**)&R_d, tam_imag*sizeof(int));

//******matriz G host y device********

G_h=(int *)malloc(sizeof(int)*tam_imag);
cudaMalloc((void**)&G_d, tam_imag*sizeof(int));

//******matriz B host y device********

B_h=(int *)malloc(sizeof(int)*tam_imag);
cudaMalloc((void**)&B_d, tam_imag*sizeof(int));

//******matriz topografica device******

T_h=(int *)malloc(sizeof(int)*tam_imag);
cudaMalloc((void**)&T_d, tam_imag*sizeof(int));

//**********matrices finales device*********

cudaMalloc((void**)&Rf, tam_imag*sizeof(int));
cudaMalloc((void**)&Gf, tam_imag*sizeof(int));
cudaMalloc((void**)&Bf, tam_imag*sizeof(int));

//******matrices Resultados finales******

R=(int *)malloc(sizeof(int)*tam_imag);
G=(int *)malloc(sizeof(int)*tam_imag);
B=(int *)malloc(sizeof(int)*tam_imag);

//**********cálculo del tiempo de procesamiento**********

float t;
clock_t tinicio, tfinal;
tinicio=clock();

        //******matriz topografica**********
        cont=0;
```

```
        for(i=0;i<dimx;i++)
                for(j=0;j<dimy;j++){
                        fscanf(file, "%d", &T_h[cont]);
                        cont++;
        }
        fclose(file)
//******operaciones*******

                for(d=1;d<=N;d++){

                //*******matriz R*********

            char ruta1[]="MiTesis/";
            sprintf(ruta1, "%s%d%s","RGB/",d,"/R");
            Red=fopen(ruta1,"r+");

            for(i=0;i<dimx*dimy;i++)
                    fscanf(Red, "%d", &R_h[i]);
            fclose(Red);

                //*******matriz G*********

            char ruta2[]="MiTesis/";
            sprintf(ruta2, "%s%d%s","RGB/",d,"/G");
            Green=fopen(ruta2,"r+");

            for(i=0;i<dimx*dimy;i++)
                    fscanf(Green, "%d", &G_h[i]);
            fclose(Green);

                //*******matriz B*********

           char ruta3[]="MiTesis/";
           sprintf(ruta3, "%s%d%s","RGB/",d,"/B");
           Blue=fopen(ruta3,"r+");

                for(i=0;i<dimx*dimy;i++)
                        fscanf(Blue, "%d", &B_h[i]);
                fclose(Blue);


                //********copia de variables del Host al Device**********

cudaMemcpy(R_d,R_h,sizeof(int)*tam_imag,cudaMemcpyHostToDevice);
cudaMemcpy(G_d,G_h,sizeof(int)*tam_imag,cudaMemcpyHostToDevice);
cudaMemcpy(B_d,B_h,sizeof(int)*tam_imag,cudaMemcpyHostToDevice);
```

```
cudaMemcpy(T_d,T_h,sizeof(int)*tam_imag,cudaMemcpyHostToDevice);
                //******llamado del kernel********

                Kernel<<<12288,400>>>(R_d, G_d, B_d, T_d, Rf, Gf, Bf, d);

        }//Fin for
                //copia de variables del Device al Host

                cudaMemcpy(R, Rf, sizeof(int)*tam_imag,
cudaMemcpyDeviceToHost);
                  cudaMemcpy(G, Gf, sizeof(int)*tam_imag,
cudaMemcpyDeviceToHost);
                cudaMemcpy(B, Bf, sizeof(int)*tam_imag,
cudaMemcpyDeviceToHost);

                //almacenamiento de las matrices resultantes*******

                ArchivoR=fopen("Resultados/R","w+");
                 ArchivoG=fopen("Resultados/G","w+");
                ArchivoB=fopen("Resultados/B","w+");

                for(i=0;i<tam_imag;i++){
                        if(i%dimy==0 && i!=0){
                                fprintf(ArchivoR,"\n");
                         fprintf(ArchivoG,"\n");
                                fprintf(ArchivoB,"\n");
                   }
                        fprintf(ArchivoR,"%d ",R[i]);
                    fprintf(ArchivoG,"%d ",G[i]);
                        fprintf(ArchivoB,"%d ",B[i]);
             }
                fclose(ArchivoR);
            fclose(ArchivoG);
                fclose(ArchivoB);

tfinal=clock();
t = ((float)tfinal-(float)tinicio)/CLOCKS_PER_SEC;
printf("tiempo de procesamiento:%6.3f s\n",t);

}//Fin función main()
```

ANEXO G: Código matlab para graficar una imagen focalizada a partir de las componentes RGB.

Código matlab para graficar una imagen focalizada a partir de las componentes RGB.

Código matlab

```
Reed=textread('R');
Green=textread('G');
Blue=textread('B');
tam=size(Reed);

I2=zeros(tam(1),tam(2),3);
I2(:,:,1)=Reed;
I2(:,:,2)=Green;
I2(:,:,3)=Blue;
I4=uint8(I2);
imshow(I4);
```

ANEXO H: Artículo presentado a la Conferencia Latinoamericana de Computación de Alto Rendimiento CLCAR.

Implementación del Método de Profundidad de Campo Extendida en Arquitecturas Paralelas

Implementing Extended Depth of Field on Parallel Architectures

M. L. Hernández[1,2], L. Camargo F.[1,2], A. Lobo[1,2], C. J. Barrios Hernández[1,2], A. Plata Gómez[1,3] y D. A. Sierra[1,2,4]

1. Universidad Industrial de Santander, UIS, 2. Computación Científica y de Alto Rendimiento 3. Grupo de Óptica y Tratamiento de Señales UIS, 4. Grupo de Investigación en Control, Electrónica, Modelado y Simulación

http://www.uis.edu.co http://sc3.uis.edu.co

Abstract

Extended depth of field (EDF) is a specific method used to analyze and treat specific image zones in optical research. Due to the complexity of the EDF and the possible large volume of data processed in optics problems, EDF is a good candidate to process in parallel architectures. This work is a first approach of implementation of parallel-extended depth of field. We propose a first solution algorithm addressed to two main architectures: distributed memory represented by a multicomputer cluster and shared memory represented by a massive parallel machine based on GPU computing. Moreover, a performance evaluation in terms of execution time is proposed followed by a discussion about this first approach.

1. Introducción

El método de profundidad de campo extendida EDF es empleado en óptica con el fin de resaltar zonas de una imagen. Al disminuir la distancia en orden de micrones, entre la imagen y el plano de foco y desplazando este último a lo largo y ancho de la imagen, se obtiene un conjunto de imágenes que se encuentran focalizadas en diferentes puntos. Para obtener una imagen completamente focalizada, se calcula la mayor variación de intensidad de color para cada pixel, haciendo una comparación para la misma posición en todas las imágenes. Este proceso requiere capacidad de cómputo y almacenamiento para una gran cantidad de datos, por lo que se propone el uso de dos arquitecturas paralelas como alternativas para realizar los cálculos: una máquina multicomputadora, Clúster Beowulf [1] que permite el análisis de multiprocesamiento usando CPU's, utilizando paso de mensajes (*MPI-message passing interface*) [2] y una máquina masivamente paralela, compuesta por GPU's, aprovechando CUDA® [3][4] (*Compute Unified Device Architecture*) una arquitectura creada por Nvidia®[5] para la programación de GPUs (*Graphics Processing Unit*) [6]. La GPU es utilizada principalmente para realizar procesamiento gráfico pero debido a su capacidad de cómputo también se emplea en aplicaciones GPGPU (*General-Purpose Computing on Graphics Processing Units*) [7] como

un procesador paralelo a la CPU, permitiendo así repartir la carga computacional del procesador central.

Este artículo está organizado como sigue: Primero se hace el planteamiento del problema, posteriormente se muestra y explica el algoritmo que se propone como solución al problema, luego se muestran las pruebas y análisis de resultados, finalmente las conclusiones a partir de las pruebas realizadas y el trabajo en curso.

2. Planteamiento del Problema

La microscopia digital, mediante el estudio y análisis de cuerpos, hace posible obtener imágenes de resoluciones de decimas de micras. Sin embargo, esta se hace sobre campo de observación de micras y no focalizada en todos sus puntos, de manera que para obtener una imagen focalizada en todo el campo es necesario realizar métodos artificiales que permitan a partir de un apilado de imágenes extraer la intensidad en la posición focalizada y la posición en la cual logro focalizarse. Además, el campo de observación es de décimas de micras y no permite tener una idea de la muestra que se observa, por tanto se necesita realizar un empalme transversal de diferentes campos de observación para obtener una imagen artificial de centésimas de micras.

Por ello se hace necesario encontrar una solución para manipular imágenes compuestas no focalizadas del orden de más de 100 mega-bytes sobre apilados de más de 100 imágenes, implementando arquitecturas que permitan mediante el uso de un método artificial almacenar y procesar imágenes microscópicas a partir de las cuales se pueda obtener una imagen de rango con resoluciones de decimas de micras en un campo amplio y además arroje una imagen focalizada, permitiendo obtener resultados acordes a la realidad.

Entonces, en una primera aproximación, se propone un algoritmo que se presenta a continuación.

3. EDF Paralelo

Se plantea la creación de un algoritmo que emplee el método de profundidad de campo extendida y el cálculo de las variaciones de intensidad de color en cada uno de los pixeles que componen una imagen. Se busca obtener un *stack* que consiste en un conjunto de imágenes del mismo objeto, pero que se encuentran focalizadas en diferentes puntos del plano de observación. Al desplazar el plano del foco a través de la imagen con una distancia de focalización en orden de micrones es posible obtener imágenes donde el campo de observación estará en centésimas de micras, al realizar un empalme de todo el *stack* como resultado se obtendrá una imagen focalizada con un mayor grado de detalle y nitidez que una imagen donde el plano de foco sea igual al campo de observación.

3.1. Algoritmo propuesto

Iniciar
Declaración de variables

Para N° imágenes de 1 a Total de imágenes hacer
Leer imagen
Obtener las componentes RGB para cada pixel→ Almacenar los datos en 3

archivos R, G, B respectivamente
Lectura de la matriz de la componente G
Calculo en paralelo de la varianza para cada posición X de la matriz G
Almacenar resultados de varianzas calculadas
Fin Para

Lectura de la primera matriz de varianza →almacenarla en un temporal
Para matriz de varianza (m) de 2 a Total de matrices
Lectura de la matriz de varianza m
Si Valor de la matriz m en la posición X > valor de temporal en la posición X
Almacenar la posición de la matriz m dentro del stack en un archivo de posiciones
Fin si
Fin Para

Lectura del archivo que contiene las posiciones en donde se encontró mayores varianzas

Para elementos de 1 a tamaño de la matriz
Lectura de la posición de cada elemento
Recuperar los valores de cada matriz R, G, B en la posición del elemento
Fin Para

Reconstruir la imagen focalizada a partir de las componentes RGB para cada pixel
Fin Algoritmo

En el algoritmo se plantean instrucciones que pueden ser procesadas en paralelo, como el cálculo de la varianza para cada pixel, en el caso de CUDA®[3][4] se propone calcular la varianza de todas las posiciones de la matriz G en paralelo. CUDA®[3][4] permite programar una GPU[6] como una malla compuesta por bloques, donde cada bloque a su vez está compuesto por hilos, de manera que, el cálculo de varianza de cada pixel puede ser asignado a cada hilo de la malla, como una operación atómica. En el caso de MPI, se propone enviar a cada nodo de trabajo matrices completas de la componente verde de la imagen tratada, para que el nodo calcule las varianzas de todos los puntos. El objetivo de tratar el algoritmo de maneras diferentes para MPI [2] y CUDA® [3][4], es que si cada nodo del clúster calcula solo la varianza de una posición, se sacrifica tiempo en paso de información a procesar de maestro-nodo y paso de resultados del nodo-maestro, para que este último reúna la información que cada nodo envía y la muestre de manera organizada. Además, se tendría que contar con un clúster compuesto por una cantidad de equipos igual a la cantidad de elementos de la matriz, lo que haría del algoritmo una solución ineficiente.

El cálculo de la varianza de cada pixel se realiza mediante una operación matemática que calcula promedio de los valores de los vecinos más próximos, al realizar una comparación con las varianzas de las demás imágenes del *stack* para un mismo punto, es posible encontrar las zonas focalizadas de una imagen, donde, una mayor variación de los promedios en dicho punto implica un área focalizada.

El algoritmo está compuesto por instrucciones que se ejecutan sobre diferentes grupos de datos, que pueden ser tratadas en paralelo, y otras que no debido a que son dependientes o que por su sencillez no tendría sentido, pues

pueden ser procesadas en la CPU. Aquellas que pueden ser ejecutadas simultáneamente serán lanzadas en el *Device* (GPU [6] y su memoria), las restantes serán lanzadas en el *Host* (CPU y su memoria) quien se encarga también de almacenar los datos del *Device* una vez han sido procesados por este último y pasados por referencia al *Host*.

Como se pretende analizar todo el plano de observación, habrá coincidencia de vecinos para el cálculo de la varianza de diferentes puntos, sin embargo, esta dependencia no impide abarcar el problema desde el ámbito de la programación en paralelo, bajo el paradigma de *Single Instruction Multiple Data* (SIMD) que emplea CUDA® [3][4], puesto que los datos originales no serán modificados y esta dependencia no está dada en tiempo. Unos primeros resultados de las pruebas realizadas al algoritmo son mostrados a continuación.

4. Pruebas y Análisis de Resultados

Las pruebas realizadas se llevaron a cabo en un clúster de computadoras y en una tarjeta GPU en el Laboratorio de Supercomputación y Cálculo Científico de la UIS. El clúster utilizado está compuesto por 5 nodos de trabajo, con Sistema Operativo Red Hat 4.1.2-42, procesador Dual-core Pentium 4, CPU 3.20 GHz, memoria RAM 2GB. La tarjeta gráfica corresponde a una tarjeta Nvidia® Quadro® FX 570, con una memoria de 256MB, instalada en un equipo que posee las mismas características de los equipos que conforman el clúster.

Las pruebas se centraron en el conjunto de instrucciones del algoritmo, encargadas de hacer los cálculos sobre la matriz G para obtener la matriz de varianza. La matriz de varianza es una matriz de decisión, que permite encontrar las zonas con mayor focalización de una determinada imagen.

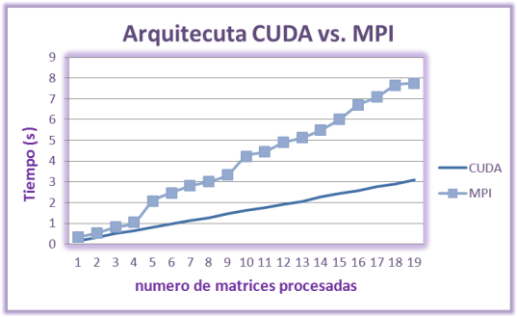
Se definieron 19 matrices que corresponden a la componente verde de 19 imágenes, de dimensiones 640*480 elementos cada una. Para facilitar el proceso de comprobar resultados, a cada elemento de las diferentes matrices le es asignado un valor de 1.
Las pruebas se limitaron a una cantidad de 19 matrices, debido a que, al contar con un clúster de 5 nodos y para evitar *overlapping*, a cada nodo se le pueden asignar 4 procesos, es decir, se cuenta con un total de 20 procesos. Para el algoritmo programado en MPI [2] se define un proceso como el proceso maestro, que está encargado de enviar información y recibir los resultados de los nodos, por tanto, el proceso maestro no realiza cálculos sobre las matrices G.

Al contar con un número de nodos inferior al número de matrices a procesar, para el algoritmo planteado para la arquitectura clúster, MPI[2] debe asignar a cada proceso varias matrices, de manera que, estos procesos deben calcular las matrices de varianza una a la vez.

Los resultados obtenidos son presentados en las Gráficas 1 y 2.

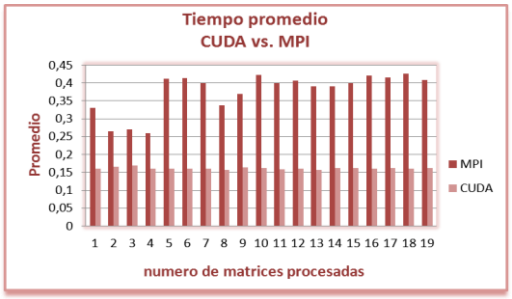
La gráfica 1 muestra una comparación de los tiempos de ejecución (segundos) el algoritmo en CUDA® [3][4] y en MPI [2] para un número de matrices de 1 a 19. Como se observa, existe un

crecimiento lineal del tiempo de ejecución en la medida que se aumenta la cantidad de matrices procesadas. Es interesante observar que existe una diferencia de un factor de 6, para la máxima cantidad de matrices. Esto se debe a las ventajas que tiene el procesamiento de problemas de grano fino en CUDA® [3][4], debido a que por un lado la arquitectura permite distribuir las matrices más eficientemente sobre cada una de las GPUs [6] y por otro lado el tiempo de comunicación es muy bajo, debido a la arquitectura misma de la tarjeta Quadro FX 570 [8].



Gráfica 1. Tiempos de ejecución CUDA® vs. MPI

En la gráfica 2 se presenta una comparación del tiempo promedio que emplea el algoritmo en CUDA® y en MPI, como resultado de dividir los tiempos de ejecución (Gráfica 1.) entre el número de matrices procesadas. La diferencia que se observa sin duda es debido a los costos de comunicación generados en MPI. Esto se debe al uso de las barreras en MPI y al costo de comunicación que es alto, comparado con CUDA®.



Gráfica 2. Tiempos promedios por matriz CUDA® vs. MPI

Estos test nos permiten llegar a algunas conclusiones realizadas a partir de los resultados obtenidos para los dos tipos de arquitecturas empleadas en este trabajo.

5. Conclusiones y Trabajo en Curso

Al realizar el cálculo de 19 matrices de varianza de dimensiones 640 filas* 480 columnas, en los dos tipos de arquitecturas, se comprobó que para el algoritmo propuesto, la arquitectura CUDA® permite tener un mayor rendimiento, en cuanto al tiempo de ejecución respecto al clúster, reduciendo los tiempos empleados por el clúster a menos de la mitad (Gráfica 1.).

Los tiempos de ejecución del algoritmo en la arquitectura CUDA®, se ven reducidos gracias a que, en cada lanzamiento del algoritmo se procesa una matriz G completa, además, los tiempos promedios que emplea esta arquitectura, para procesar un número *n* de matrices, tiende a ser constante, lo que permite comprobar que CUDA® presenta latencias bajas (Gráfica 2.).

Debido a que con MPI se hace envío de información a través de la red, cuando los datos son de gran volumen, se presenta retardos en la comunicación y

por ende un aumento en el tiempo de ejecución.

Aunque en realidad estamos trabajando sobre una plataforma híbrida, GPU-CPU que implica multicomputadoras y una máquina masivamente paralela que reúne características de memoria compartida y memoria distribuida, se comprobó, que para el algoritmo planteado, la arquitectura CUDA® es la mejor opción en términos de tiempo de ejecución, pues ofrece un mejor rendimiento en cuanto a tiempo de procesamiento para un número determinado de matrices, debido a la granularidad del problema.
Actualmente se trabaja en el desarrollo y evaluación de algoritmos que aprovechen arquitecturas híbridas, es decir, aquellas que impliquen procesamiento en GPUs y en CPUs, aprovechando las características arquitecturales de las plataformas, aunque implica más complejidad en la implementación.

6. Agradecimientos

Los autores expresan sus agradecimientos especiales al Dr. Oscar Gualdrón González, vicerrector de investigación y extensión de la UIS.

Los resultados de los experimentos presentados en esta publicación fueron obtenidos usando la plataforma GridUIS-2, desarrollada por el Servicio de Cómputo de Alto Rendimiento y Calculo Científico de la Universidad Industrial de Santander (UIS). Esta acción es soportada por la Vicerectoría de Investigación y Extensión de la UIS (VIE-UIS) y diferentes grupos de investigación de la universidad. (http://sc3.uis.edu.co).

7. Referencias

[1] Wittwer Tobias, An Introduction to Parallel Programming, VSSD, Leeghwaterstraat 42, 2628 CA Delft, The Netherlands, (First edition 2006).

[2]George Em Karniadakis and Robert M. Kirby II, Parallel Scientific Computing in C++ and MPI, A seamless approach to parallel algorithms and their implementation, Cambridge University Press, Cambridge, Uneted Kingdom, First published 2003.

[3] Jason Sanders, Edward Kandrot, CUDA by Example An Introduction to General-Purpose GPU Programming, Nvidia, Addison Wesley, Ann Arbor, Michigan, United States, First printing July 2010.

[4]NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 1.1, Nvidia Corporation, 11/29/2007.

[5] Nvidia Web Site. http://www.nvidia.com

[6] Nvidia. GPU Computing. http://www.nvidia.es/page/gpu_computing.html

[7] GPGPU General-Purpose computation on Graphics Processing Units Web Site. http://gpgpu.org.

[8] Nvidia Quadro Family Tech Specs http://www.nvidia.fr/page/quadrofx_family.html

ANEXO I: Script en Bash para ejecutar los 3 segmentos del algoritmo

```
#!/bin/bash
echo  -e "\n"
echo  -e "\t\tUNIVERSIDAD INDUSTRIAL DE SANTANDER"
echo  -e "\t\t\tPROYECTO DE GRADO:"
echo  -e "\t\t\t\b\b\bTRATAMIENTO DE IMÁGENES"
echo  -e " \tEMPLEANDO ELMÉTODO DE PROFUNDIDAD DE CAMPO
EXTENDIDA"
echo  -e "\t\t\t\b\b\b\bEN ARQUITECTURAS PARALELAS"
echo  -e "\n"
echo  -e "\t\t  MÓNICA LILIANA HERNÁNDEZ ARIZA"
echo  -e "\n"

./cudaV $1 $2
./cudaT $1 $2
./cudaRGB $1 $2
```